

FREENIX Track
2002 USENIX Annual
Technical Conference

Monterey, California, USA
June 10–15, 2002

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$22 for members and \$30 for nonmembers.
Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past FREENIX Proceedings

FREENIX '01	2001	Boston, MA	\$22/30
FREENIX '00	2000	San Diego, CA	\$22/30
FREENIX '99	1999	Monterey, CA	\$22/30

© 2002 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-01-4

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.

USENIX Association

**Proceedings of the
FREENIX Track**

2002 USENIX Annual Technical Conference

**June 10–15, 2002
Monterey, California, USA**

Program Organizers

Program Chair

Chris Demetriou, *Broadcom Corp.*

Program Committee

Chuck Cranor, *AT&T Labs—Research*

Jim McGinness, *Consultant*

Craig Metz, *Extreme Networks*

Toon Moene, *GNU Fortran Team*

Keith Packard, *XFree86 Core Team & Compaq Computer Corp.*

Niels Provos, *University of Michigan*

Robert Watson, *NAI Labs & The FreeBSD Project*

Erez Zadok, *Stony Brook University*

The USENIX Association Staff

External Reviewers

Clem Cole, *Paceline Systems*

Michael J. Leibensperger, *Paceline Systems*

Chuck Lever, *Network Appliance*

Jeffrey Mogul, *Compaq Western Research Lab*

Jason Nieh, *Columbia University*

Amit Purohit, *Stony Brook University*

Joseph Spadavecchia, *Stony Brook University*

Thomas Teixeira, *Acopia Networks*

2002 USENIX Annual Technical Conference
FREENIX Track
June 10–15, 2002
Monterey, CA, USA

Index of Authors	vii
Message from the Program Chair	ix

Thursday, June 13

Building Applications

Session Chair: Chris Demetriou, Broadcom Corp.

Interactive 3D Graphics Applications for Tcl	1
<i>Oliver Kersting and Jürgen Döellner, Hasso Plattner Institute for Software Systems Engineering, University of Potsdam</i>	

The AGFL Grammar Work Lab	13
<i>Cornelis H.A. Koster and Erik Verbruggen, University of Nijmegen (KUN)</i>	

SWILL: A Simple Embedded Web Server Library	19
<i>Sotiria Lampoudi and David M. Beazley, University of Chicago</i>	

Network Performance

Session Chair: Craig Metz, Extreme Networks

Linux NFS Client Write Performance	29
<i>Chuck Lever, Network Appliance, Incorporated; and Peter Honeyman, CITI, University of Michigan</i>	

A Study of the Relative Costs of Network Security Protocols	41
<i>Stefan Miltchev and Sotiris Ioannidis, University of Pennsylvania; and Angelos Keromytis, Columbia University</i>	

Congestion Control in Linux TCP	49
<i>Pasi Sarolahti, University of Helsinki; and Alexey Kuznetsov, Institute for Nuclear Research at Moscow</i>	

Xtreme Xcitement

Session Chair: Keith Packard, XFree86 Core Team & Compaq Computer Corp.

The Future Is Coming: Where the X Window System Should Go	63
<i>Jim Gettys, Compaq Computer Corporation</i>	

XCL: An Xlib Compatibility Layer for XCB	71
<i>Jamey Sharp and Bart Massey, Portland State University</i>	

Biglook: A Widget Library for the Scheme Programming Language	85
<i>Erick Galliesio, University of Nice – Sophia Antipolis; and Manuel Serrano, INRIA Sophia Antipolis</i>	

Friday, June 14

Hacking in the Kernel

Session Chair: Chuck Cranor, AT&T Labs—Research

An Implementation of Scheduler Activations on the NetBSD Operating System 99
Nathan J. Williams, Wasabi Systems Inc.

Authorization and Charging in Public WLANs Using FreeBSD and 802.1x 109
Pekka Nikander, Ericsson Research NomadicLab

ACPI Implementation on FreeBSD 121
Takanori Watanabe, Kobe University

Analyzing Applications

Session Chair: Jim McGinness, Consultant

Gscope: A Visualization Tool for Time-Sensitive Software 133
Ashvin Goel and Jonathan Walpole, Oregon Graduate Institute, Portland

Inferring Scheduling Behavior with Hourglass 143
John Regehr, University of Utah

A Decoupled Architecture for Application-Specific File Prefetching 157
Chuan-Kai Yang, Tulika Mitra, and Tzi-Cker Chiueh, Stony Brook University

Access Control

Session Chair: Robert Watson, NAI Labs & The FreeBSD Project

Design and Performance of the OpenBSD Stateful Packet Filter (pf) 171
Daniel Hartmeier, Systor AG

Enhancing NFS Cross-Administrative Domain Access 181
Joseph Spadavecchia and Erez Zadok, Stony Brook University

Saturday, June 15

Engineering Open Source Software

Session Chair: Niels Provos, University of Michigan

Ningau: A Linux Cluster for Business 195
Andrew Hume, AT&T Labs—Research; and Scott Daniels, Electronic Data Systems Corporation

CPCMS: A Configuration Management System Based on Cryptographic Names 207
Jonathan S. Shapiro and John Vanderburgh, Johns Hopkins University

X Meets Z: Verifying Correctness in the Presence of POSIX Threads 221
Bart Massey, Portland State University; and Robert T. Bauer, Rational Software Corporation

File Systems

Session Chair: Erez Zadok, Stony Brook University

Planned Extensions to the Linux Ext2/Ext3 Filesystem 235

Theodore Y. Ts'o, IBM; and Stephen Tweedie, Red Hat

Recent Filesystem Optimisations on FreeBSD 245

Ian Dowse, Corvil Networks; and David Malone, CNRI Dublin Institute of Technology

Filesystem Performance and Scalability in Linux 2.4.17 259

Ray Bryant, SGI; Ruth Forester, IBM; and John Hawkes, SGI

Things to Think About

Session Chair: Toon Moene, GNU Fortran Team

Speeding Up Kernel Scheduler by Reducing Cache Misses 275

Shuji Yamamura, Akira Hirai, Mitsuru Sato, Masao Yamamoto, Akira Naruse, and Kouichi Kumon, Fujitsu Laboratories, LTD

Overhauling Amd for the '00s: A Case Study of GNU Autotools 287

Erez Zadok, Stony Brook University

Simple Memory Protection for Embedded Operating System Kernels 299

Frank W. Miller, University of Maryland, Baltimore County

Index of Authors

Bauer, Robert T.	221	Malone, David	245
Beazley, David M.	19	Massey, Bart	71, 221
Bryant, Ray	259	Miller, Frank W.	299
Chiueh, Tzi-Cker	157	Miltchev, Stefan	41
Daniels, Scott	195	Mitra, Tulika	157
Doellner, Juergen	1	Naruse, Akira	275
Dowse, Ian	245	Nikander, Pekka	109
Forester, Ruth	259	Regehr, John	143
Gallesio, Erick	85	Sarolahti, Pasi	49
Gettys, Jim	63	Sato, Mitsuru	275
Goel, Ashvin	133	Serrano, Manuel	85
Hartmeier, Daniel	171	Shapiro, Jonathan S.	207
Hawkes, John	259	Sharp, Jamey	71
Hirai, Akira	275	Spadavecchia, Joseph	181
Honeyman, Peter	29	Ts'o, Theodore Y.	235
Hume, Andrew	195	Tweedie, Stephen	235
Ioannidis, Sotiris	41	Vanderburgh, John	207
Keromytis, Angelos	41	Verbruggen, Erik	13
Kersting, Oliver	1	Walpole, Jonathan	133
Koster, Cornelis H.A.	13	Watanabe, Takanori	121
Kumon, Kouichi	275	Williams, Nathan J.	99
Kuznetsov, Alexey	49	Yamamoto, Masao	275
Lampoudi, Sotiria	19	Yamamura, Shuji	275
Lever, Chuck	29	Yang, Chuan-Kai	157

Message from the Program Chair

FREENIX is a special track within the USENIX Annual Technical Conference where people working with open-source software are encouraged to share the results of their work with others. We have especially tried to encourage authors with little or no publishing experience, and to provide a premier venue in which they can present their work. Our hope is that their knowledge and experience will benefit others in the computer science community.

These proceedings of the FREENIX track of the 2002 USENIX Annual Technical Conference describe some of the most exciting recent and ongoing work in the open-source software community.

For the third year running, the FREENIX program was selected using a formal peer-review process and every presentation has a corresponding paper in these proceedings. We received 53 submissions, ranging from two-page summaries to full-length papers. The program committee and external reviewers read, reviewed, and provided detailed feedback on those submissions. Ultimately, we produced over 13,000 lines—200 pages—of review comments, which were provided to the authors to help them improve their papers and their work.

At the program committee meeting in January 2002, the committee worked day and night (literally, one of each!) to select the submissions that would ultimately become this year's FREENIX program. We selected the best submissions, organized them into conference sessions, and assigned each submission a "shepherd" responsible for helping the authors turn their work into a paper ready for publication.

From January until mid-April, the authors produced, and the shepherds read and reviewed, countless additional paper drafts. Authors of submissions were not simply expected to expand those submissions to fill additional space; with the help of their shepherds, they were required to address the issues and comments raised by the committee and the reviewers, and to produce papers both easily readable and worth reading. For some papers, a dozen drafts or more were exchanged before the papers were finished. In mid-April, with the 26 final papers in hand, the USENIX Association staff produced these proceedings.

It's been my great pleasure to work with the authors, program committee members, external reviewers, and USENIX Association staff members to create this year's FREENIX program. I'd like to thank them all for their months and years of hard work, without which the FREENIX track would not be possible, or would not meet the high standards that we've come to expect. I would especially like to thank Erez Zadok, Clem Cole, and Jane-Ellen Long for their encouragement, assistance, patience, and perseverance.

I strongly encourage you to read these proceedings and to spend some time in the FREENIX sessions at the Annual Technical Conference this year in Monterey or in the future. One of the great strengths of both the USENIX Association and the open-source software development community is that they encourage communication and interaction among individuals of diverse backgrounds and interests. You are welcome and encouraged to participate: meet the authors and speakers, discuss their work and yours, and exchange ideas. Such exchanges drive the success of open-source software and of conferences such as FREENIX; you might discover opportunities for collaboration, or even consider writing a future paper yourself.

Chris Demetriou, Program Chair

Interactive 3D Graphics for Tcl

Oliver Kersting and Jürgen Döllner

kersting@hpi.uni-potsdam.de, doellner@hpi.uni-potsdam.de

*Hasso Plattner Institute for Software Systems Engineering
University of Potsdam*

Abstract

This paper presents an approach to integrate interactive real-time 3D graphics into the scripting language Tcl. 3D graphics libraries are typically implemented in system programming languages such as C or C++ in order to be type safe and fast. We have developed a technique that analyzes the C++ application programming interface of such a library and maps it to appropriate scripting commands and structures. As 3D graphics library, we apply the Virtual Rendering System, an object-oriented library that supports 3D modeling, interaction, and animation. The mapped API represents a complete and powerful development tool for interactive, animated 3D graphics applications. The mapping technique takes advantage of the weak typing and dynamic features of the scripting language, preserves all usability-critical features of the C++ API, and has no impact on performance so that even real-time 3D applications can be developed. The mapping technique can be applied in general to all kinds of C++ APIs and automated. It also gathers reflection information of the API classes and supports interactive management of API objects. Consequently, interactive development environments can be built easily based on this information. We illustrate the approach by several examples of 3D graphics applications.

1 Introduction

In 3D computer graphics and multi-media, a large number of software libraries have been developed in the past. Examples include the 3D graphics library OpenInventor [15], the visualization toolkit VTK [13], the 3D rendering system OpenGL [16], and an upcoming standard for media creation and playback called OpenML [7]. As a common characteristic, these libraries provide low-level C/C++ application programming interfaces (APIs) in terms of functions, data types, and classes. Developing applications based on these APIs is generally regarded difficult because developers are frequently confronted with a multitude of functionality and the implications of strong typing, and typically it becomes difficult to explore and use the library's capabilities.

Scripting languages have been successfully used to develop large, complex software systems [11]. The reasons for this include: 1) Scripting languages simplify gluing together existing software components to a single application. The simplification results mainly from weak typing because this allows developers to interface components that use incompatible data types at the programming-language level. 2) An API of a pre-built library can be integrated by adding new commands to the scripting language. Such an extension to a scripting language is called *binding* of an API. A binding further simplifies the application development because it gives developers easy access to the API and does not require detailed knowledge of the underlying system programming language. 3) Scripting enables developers to experiment with libraries and their functionality in a quite different way compared to studying a C++ API documentation. Interactively, developers can

- instantiate objects and see immediately whether their services fit to the problem to be solved;
- inquire available classes, methods, method arguments, and other public class elements;
- inquire current objects and manage these objects;
- modify and deploy objects; and
- easily integrate object management and object visualization into the graphical user interface.

In this paper, we discuss how to bind the C++ API of the Virtual Rendering System VRS [3], a complex and large library for real-time 3D computer graphics, to the scripting language Tcl [10]. The resulting Tcl/Tk package is called *interactive VRS* (iVRS). The *mapping*, which denotes the process of generating the binding, is based on an automated analysis of the class interfaces and generation of appropriate wrapper classes. The binding, however, must guarantee that a) all features of the API are transparently accessible in the scripting language and b) no major performance penalties are introduced. As proof-of-concept, we have developed an interactive real-time 3D-map system, called *LandExplorer*, which can be used to visualize and explore geo data (Figure 1).

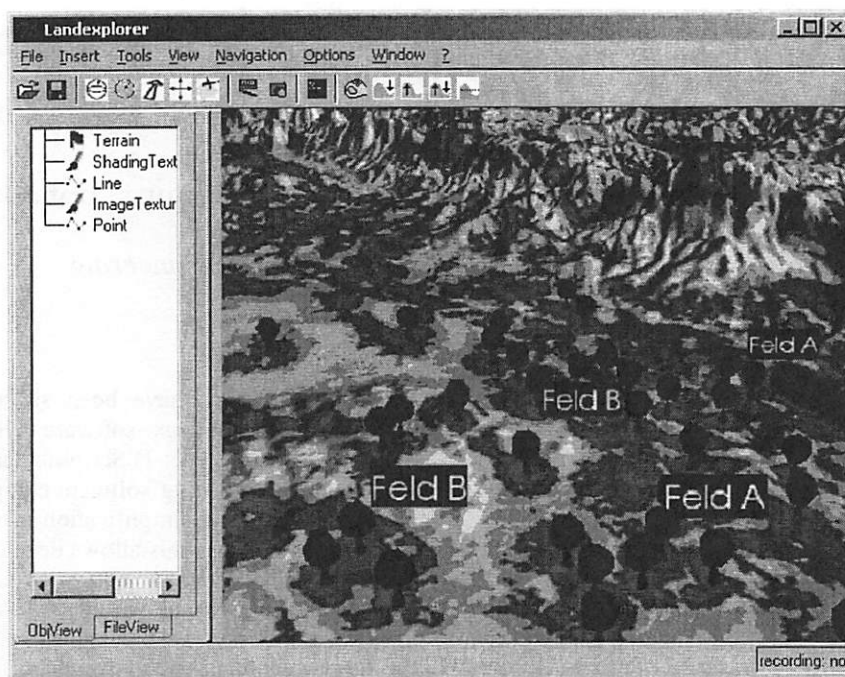


Figure 1: The interactive 3D-map system *LandExplorer*, which has been built with iVRS. LandExplorer is used to visualize, explore, and analyze geo data.

Section 2 briefly discusses related work. Section 3 outlines main aspects of the Virtual Rendering System and its API. Section 4 discusses the requirements a binding should meet to simplify application development. Section 5 explains methods for mapping C++ APIs to Tcl bindings. Section 6 gives examples that illustrate how developers can interactively use iVRS. Finally, Section 7 draws some conclusions.

2 Related Work

Tcl is basically a procedural language bundled with the powerful user-interface toolkit Tk. A couple of object-oriented extensions exist that integrate many concepts of object-oriented programming languages in Tcl (e.g., classes, encapsulation, inheritance etc.), for example [incr Tcl] [8] and the extension of Sinnige [14], which mimics C++ class syntax. Using these extensions, developers can specify and implement classes in Tcl. However, it is rather difficult to implement real-time 3D graphics on top of these extensions because graphics data has to be efficiently processed and stored.

The Tcl interpreter can be extended by new commands, which interface external functions written in C or C++; object-oriented features of C++ are not directly supported by Tcl. Therefore, several techniques and tools, which are called *mappers*, have been developed to map the API of an existing C++ class hierarchy into adequate constructs of the Tcl language. A general C++-to-Tcl mapper is SWIG, the Simplified Wrapper and Interface Generator [1]. SWIG can map C, C++, and Objec-

tive-C classes into a variety of higher-level languages (e.g., Tcl, Python) by parsing the header files or special interface files and generating wrapper code for the scripting language. One limitation is that certain C++ features are not directly mapped, e.g., overloaded methods in C++ cannot be handled without a name affix, and smart pointers are not supported. In our approach, we focus on usability-critical features of C++ APIs such as overloaded methods, object management, memory management, and class reflection.

TkOpenGL [4] directly maps the OpenGL API to Tcl. For each OpenGL C function, there is a Tcl command. This way, simple scenes can be built based on OpenGL functions, but complex scene modeling can only be achieved using large numbers of calls to OpenGL C functions via Tcl, which decreases speed drastically.

The Itcl++ mapper [5] is based on [incr Tcl]. Itcl++ parses C++ headers and generates C++ wrapper code that is linked to generated [incr Tcl] classes. For each C++ class, C++ wrapper code and a corresponding [incr Tcl] class is generated. The inheritance relationships between classes are preserved under this mapping. This way, new [incr Tcl] classes can be derived from generated [incr Tcl] classes. As a proof of concept, the authors completely wrapped the C++ API of the OpenInventor 3D graphics library. The limitations of this approach include the missing support for abstract classes and overloaded methods and the costly mapping process. Our approach targets the same category of C++ APIs but does support important C++ API features while using significantly less resources because classes

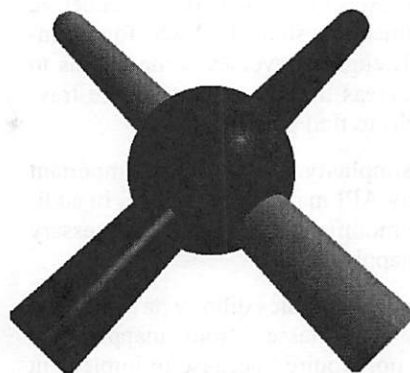
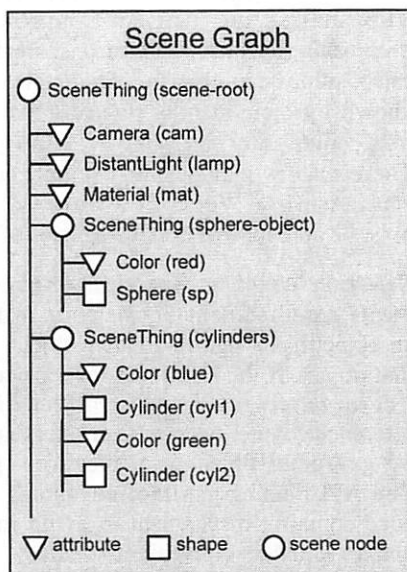


Figure 2: Example of a VRS scene graph (left) and a snapshot taken from the rendered scene (right).

are not mirrored into classes of an object-oriented extension of Tcl.

The TclObj mapper [12] works based on C++ macros and requires the original C++ source to be modified in order to access attributes and methods. As a major disadvantage of this approach, the mapping process is not automatic.

The 3D game engine Nebula [9] represents an example of a complex C++ library wrapped by Tcl focusing on game development. The Nebula library is based on the scene graph paradigm and can be extended by C++ class inheritance. Through the Tcl API, developers can construct and configure scene graphs. Nebula provides a good example of how efficiently real-time 3D games can be created by scripting languages. VRS, however, is a general-purpose 3D graphics library, intended for any kind of 3D application.

3 The 3D Graphics Library VRS

The *Virtual Rendering System* [3] is a 3D graphics library providing a large collection of 3D building blocks including different types of geometries, graphics attributes, and state-of-the-art real-time 3D rendering techniques such as shadowing, reflection, and multi-texturing. Interactive, dynamic 3D applications are implemented by scene graphs and behavior graphs. A scene graph specifies geometry and appearance of a 3D scene, whereas a behavior graph specifies event handling, time-dependent actions, and constraint handling [2]. VRS supports advanced real-time rendering techniques such as texture-based shadows, reflection, or multi-texturing capabilities and encapsulates low-level OpenGL techniques like P-buffer rendering [17].

The scene graph is composed of scene nodes, called *scene things*. The nodes basically serve as containers for *node components*. Node components include shapes (3D geometries), graphics attributes (e.g., color, material specifications, light sources), geometric transformations (e.g., rotation, scaling, translation), and child nodes, which represent scene subgraphs (Figure 2). In analogy, the behavior graph is composed of behavior nodes that contain node components. For behavior nodes, node components include constraints, time requirements, time layouts, and event handlers. To display a scene, rendering engines traverse scene graphs and interpret the node components of the nodes. To animated and interact with scenes, events are sent through and processed by behavior graphs.

The API of VRS uses common C++ language features such as classes, encapsulation, method overloading, inheritance, and templates. The API is strictly object-based and maintains encapsulation, that is, objects are exclusively accessed by method calls. VRS makes a clear distinction between API classes and internal classes (e.g., interface classes and private classes).

The VRS API can be considered as a typical API of an object-oriented graphics or multimedia library, whose usage can be characterized by the following main tasks:

- *Constructing models.* To construct scene graphs and behavior graphs, developers search for appropriate classes, look through the services they provide, and use instances of these classes to construct 3D scenes.
- *Modifying models.* To modify models, their components have to be accessed. To do so, developers want to inquire objects of a certain kind currently

instantiated, inspect their state, inquire provided services, and call methods.

- *Evaluating models.* To evaluate models, they are interpreted with respect to an underlying rendering system or multimedia system. In VRS, for example, the OpenGL engine traverses scene graphs to render them, whereas the ray-request engine traverses scene graphs to find picking results.

Completeness and simplicity are the most important quality criteria for any API mapping technique. In addition, no source code modifications should be necessary to generate an API mapping.

The ability to extend a graphics library at Tcl level (e.g., deriving new Tcl classes from mapped Tcl classes) is generally not required because to implement extensions, access to low-level functionality of the underlying system libraries (e.g., OpenGL) is necessary. When the C++ library has been extended, the mapping process can be invoked again. Developers able to extend the library will also be able to control the mapping process.

4 Requirements of API Bindings

A binding of an API should allow developers to easily access all interface elements of the C++ API within an interactive environment such as the Tcl interpreter. This way, developers can experimentally explore the API and incrementally develop applications at runtime. For this, the API binding should support creating, inspecting, modifying, and destroying objects by Tcl commands.

The API binding needs an efficient and powerful mechanism to identify and call methods of the C++ API. In particular, string arguments of Tcl commands must be automatically interpreted and converted to typed C++ arguments (and vice versa), and a corresponding method must be chosen based on the method signature. To illustrate that API mapping is subtle undertaking, let us consider a few details of the VRS API:

- *Objects with identity versus objects as values.* Most VRS classes are derived from the root class `SharedObj`, which represents shareable, dynamically allocated objects. Shareable objects are known by their identity. Objects that are handled like values are not derived from `SharedObj`. These classes include, for example, mathematical classes such as `Vector` or `Matrix`. The mapper must be able to treat both categories of objects.
- *Reference counting.* The class `SharedObj` implements a semi-automatic memory management. If an object has a link to another object, it references that object. If the link is no longer needed, the object un-references the other object. If an object's reference counter becomes zero, it is not referenced by any other object and, therefore, gets deleted. The API binding can take advantage of the reference counting mechanism to avoid memory leaks and to provide convenient memory management from a developer's point of view.
- *Method overloading.* Many VRS classes specify overloaded methods (e.g., two or more constructors). If overloaded methods differ in the number of arguments, they can clearly be distinguished. If not, the API binding must be able to distinguish the argument types although Tcl does not support argument typing.
- *Default argument values.* Methods can define default values for their arguments. The API binding should allow us to call these methods with and without default arguments.
- *Abstract classes.* The API binding must detect abstract classes and prohibit instantiating objects of them.
- *Template classes.* All VRS data container classes are implemented as template classes. The API binding must generate frequently used template classes in advance.

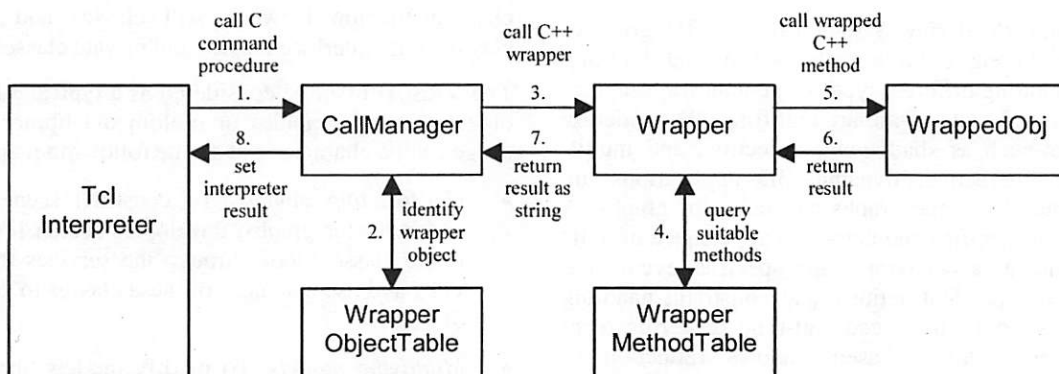


Figure 3: The handling of method calls by the Tcl binding.

- *Enumerations.* Symbolic names specified as enumeration constants are frequent elements of VRS classes. The API binding must preserve these names in its communication with the Tcl interpreter.
- *Static class elements.* Static methods are frequent elements in VRS classes. The API binding must integrate static methods similar to the syntax used in C++, that is, static methods should be accessible via the class name.

VRS consists of approximately 530 C++ classes; about 280 are relevant for being mapped to Tcl. These 280 classes include 6 numerical classes and about 45 template classes. All classes implement about 2500 methods. The 70 base classes define 54 abstract methods. Method overloading occurs 88 times.

5 Mapping Techniques

Our mapping technique concentrates on object-oriented language features of C++. It maps, as basic constructs, classes and template classes. In the case of templates, concrete instantiations must be specified. For each class, it can handle virtual and non-virtual methods, static methods, overloaded methods, and arithmetic operators. In addition, single inheritance is supported.

The mapping technique does currently not support direct access to member variables, non-constant references, namespaces, nested classes, and multiple inheritance. These language features rarely occur in the VRS API. They could be added in a straightforward way to the mapping technique.

The key idea for mapping a C++ class to the Tcl binding is to create a *wrapper class*, which collects interface

information and reflects C++ class methods with their signatures consisting of string arguments only. The mapping technique obtains interface information by parsing the C++ header files. The wrapper class uses *converter functions* to transform between strings and C++ data types, and it links wrapper methods and wrapped methods (Figure 3). Both, wrapper classes and converter functions are generated automatically.

The C++ object instantiated by the library is called *wrapped object*. It is accessed by Tcl through a corresponding *wrapper object*. The *call manager* handles VRS-related scripting commands. First, it identifies the wrapper object by the *wrapper-object table*. Next, it is looking through the *wrapper-method table* of the wrapper object for an appropriate method, called the *wrapper method*. If found, it calls that method, which converts its string arguments into C++ data types and subsequently calls the corresponding method of the wrapped object.

5.1 Wrapper Classes

Wrapper classes are responsible for type-sensitive method selection and delegation of their execution to C++ method calls. The class `Wrapper` is the common base class of all concrete wrapper classes. It defines methods for identifying the class of an object and all its base classes. The “class” of the wrapper base class is `void*`. Derived wrapper classes specialize the identification methods. For example, the wrapper class for the `Sphere` class defines that its wrapped objects belong to the `Sphere` class and its base classes `Shape` and `SharedObj`.

A wrapper method expects an array of strings as arguments. The following example (Figure 4) shows the

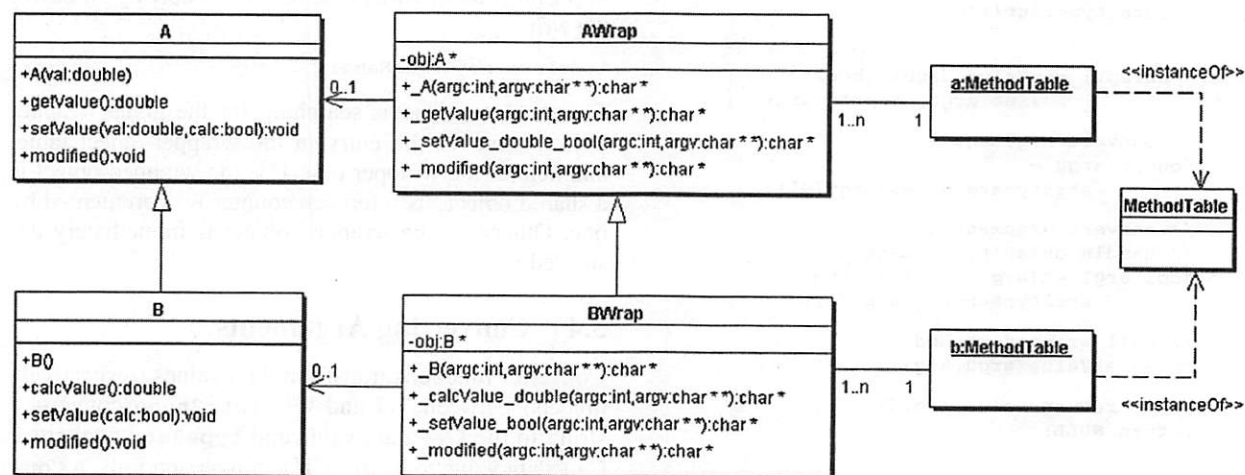


Figure 4: A sample class hierarchy (A and B), its corresponding wrapper classes (AWrap and BWrap), and their associated wrapper-method tables.

data relations among the class **A** and its wrapper class **AWrap**, implemented as follows:

```
// wrapped C++ class
class A {
    double value;
public:
    // constructor
    A(double val = 1.0);

    // set/get value
    double getValue();
    void setValue(double val,
                  bool calc = true);

    // polymorph method modified
    virtual void modified();
};

// C++ wrapper class
class AWrap : public Wrapper {
    // pointer to wrapped object
    A* obj;
public:
    // constructor wrapper method
    char* _A(int argc, char** argv);

    // set/get value wrapper method
    char* _getValue(int argc, char** argv);
    char* _setValue_double_bool(int argc,
                                char** argv);

    // modified wrapper method
    char* _modified(int argc, char** argv);
};
```

The implementation of two typical wrapper methods is outlined in the following example (converter functions are named **type2str** respectively **str2type**):

```
char* Awrap::_getValue(int argc, char** argv){
    // call wrapped method
    double res = obj->getValue();

    // convert return type to char*
    return type2str(res);
}

char* Awrap::_setValue_double_bool
(int argc, char** argv) {
    // convert argument 0
    double arg0 =
        str2type<double>(argv[0]);

    // convert argument 1
    // handle default argument
    bool arg1 = (argc <= 1) ? true :
        str2type<bool>(argv[1]);

    // call wrapped method
    obj->setValue(arg0, arg1);

    // no return value (void)
    return NULL;
}
```

5.2 Wrapper-Method Table

The wrapper-method table stores signature information for each mapped method. The signature information includes method name, arguments, minimum and maximum number of method parameters, flags, and the pointer to the wrapper method (Table 1 and 2). The flags indicate special-purpose methods such as constructors (**CTOR**) or static methods (**STATIC**). The signature information is needed for correctly resolving overloaded methods and argument default values.

The mapper creates a wrapper-method table by copying the wrapper-method table of the base class (if any), except methods marked with **CTOR** or **STATIC**. Next, it inserts entries for methods declared in the class under consideration. An entry replaces a stored entry if method name and arguments are the same. This way, overridden methods are handled. For example, see the **modified** entry in the wrapper-method table of class **B** (Figure 4).

5.3 Memory Management

The Tcl commands **new** and **delete** create and destroy, respectively, wrapped objects. The constructor command has the following format:

```
% new ClassName arg1 arg2 ...
objClassName1
```

The **CallManager** instantiates a new wrapper object depending on the class name written after **new** and iterates over all methods marked with the flag **CTOR** in the wrapper-method table. The first constructor method that can convert all incoming arguments is called and the wrapped object is constructed. If a shared object is created, its reference counter is incremented by one. The **CallManager** generates a unique name for the new object and returns it as result. To destroy an object we write:

```
% delete objClassName1
```

The **CallManager** is searching for the given wrapper object, removes the entry in the wrapper-object table, and deletes the wrapper object. If the wrapped object is a shared object, its reference counter is decremented by one. Otherwise the wrapped object is immediately destroyed.

5.4 Converting Arguments

Converter functions transform data values (respectively objects) between Tcl and C++: **str2type** converts a string to the C++ data value and **type2str** converts a C++ data value to a string. If a conversion fails, a **ConvertException** is thrown. The **CallManager** catches this exception. This way, overloaded methods are distinguished.

For standard data types (e.g., `char`, `int`, `double`) converter functions are built-in. Derived standard data types, like `double***`, are interpreted as non-typed data, that means they are treated as `void*`. Converter functions for object pointers can be handled type-safe because of the stored type information in each wrapper object. Converter functions for objects of classes derived from `SharedObj` can additionally use the C++ runtime type information and convert precisely to the underlying type. In addition, inline conversion provides a complementary handling of arguments that are handled “by-value”.

5.5 Inline Conversion

Numerical classes such as points, vectors, matrices, and rays are frequently used in graphics and multimedia applications. Numerical objects are mostly transient, i.e., applications use them by value. The example below shows how to add vectors:

```
% set v [new Vector 1 2 3]
% $v + "4 5 6"
5 7 9
% delete $v
```

To support numerical objects, we could use the same mechanism as for non-transient objects. In the example, we could initialize a new `VectorWrapper` object with the arguments “1 2 3”, register and return its name, call the method “+” with the argument “4 5 6”, deregister its name, and delete the wrapper object. This approach, however, is neither syntactically elegant nor computationally efficient.

Alternatively, we could implement all API relevant methods of numerical classes as Tcl procedures. This would duplicate the implementation, lead to less efficient implementations, and involve manual work.

To cope with transient objects handled by-value, our mapping technique supports inline conversion of numerical objects using directly transient C++ objects.

The vector example above is written as follows:

```
% VECTOR "1 2 3" + "4 5 6"
5 7 9
```

The `VECTOR` function initializes a static `VectorWrapper` object with the first argument “1 2 3”, calls the method “+” with the argument “4 5 6”, and returns the result. This way, we save time for creation, registration, deregistration, and destruction of the wrapper object. Inline conversion functions are generated automatically for numerical classes such as `Vector`, `Color`, `Matrix`, `Ray`, and `Area`.

5.6 Enumerations

Enumerations typically represent integer constants by symbolic names and, this way, facilitate the usage of these constants. An enumeration in C++ is defined as a pair of integer value and name.

Our mapping technique supports C++ enumerations similar to the C++ syntax by `classname::enumname`. An enumeration specified this way in Tcl can be directly mapped to its C++ counterpart. To map a C++ enumeration to its Tcl name, the enumeration type (i.e., the class) must be known, otherwise only the integer value of the enumeration value can be returned.

```
% set polygon [new PolygonSet
                  PolygonSet::Quads]
% $polygon getType
PolygonSet::Quads
```

5.7 Overloaded Methods

The C++ compiler can differentiate between overloaded methods at compile time based on argument number and/or argument type. The Tcl interpreter cannot differentiate based on argument types because Tcl is typeless.

To solve this problem, our mapping technique uses a try-and-error strategy. In the case of overloaded meth-

Method Name	Arguments	Min.	Max.	Flags	Method Pointers
"A"	"double"	1	1	CTOR	AWrap:: A
"setValue"	"double bool"	1	2		AWrap:: setValue double bool
"getValue"	""	0	0		AWrap:: getValue
"modified"	""	0	0		AWrap:: modified

Table 1: Wrapper-method table for class A.

Method Name	Arguments	Min.	Max.	Flags	Method Pointers
"B"	""	0	0	CTOR	BWrap:: B
"setValue"	"double bool"	1	2		AWrap:: setValue double bool
"setValue"	"bool"	1	1		BWrap:: setValue bool
"getValue"	""	0	0		AWrap:: getValue
"modified"	""	0	0		BWrap:: modified
"calcValue"	""	0	0		BWrap:: calcValue

Table 2: Wrapper-method table for class B.

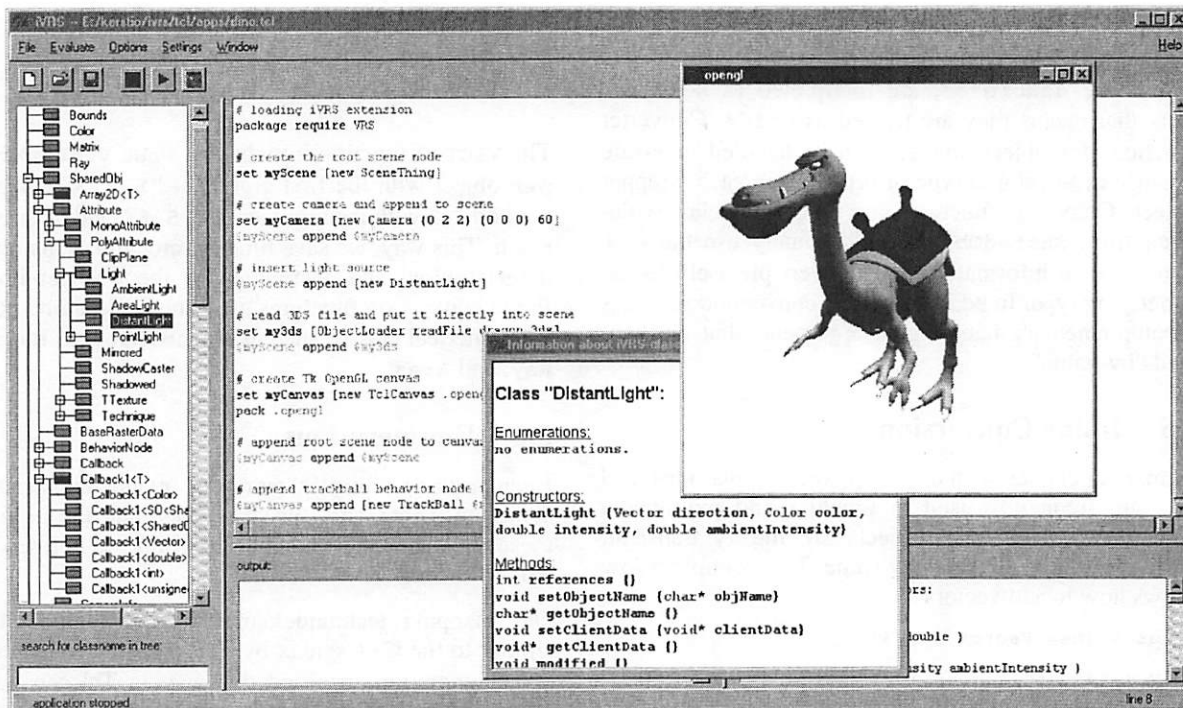


Figure 5: Integrated development environment for iVRS.

ods, we iterate through the method table searching for methods with the specified name and the correct number of arguments. For each suitable method, we try to convert argument strings into C++ data objects. If a conversion exception occurs, the next suitable method will be searched. If no exception occurs, the method call was successful. The method-table entry of a successful method is moved to the beginning of the table to reduce the number of method tests for the next call of the same method.

Some data types, however, cannot be distinguished by their value. For example, the characters 17 can be interpreted as `int`, `float`, `char`, or `string`. In such a case, the `CallManager` calls the *first* method that can convert all arguments of a method. To avoid this non-deterministic behavior, the name of the method can explicitly force a specific data conversion. Example:

```
$obj setValue:char 17
```

or force conversion as string

```
$obj setValue:string 17.
```

If no method with the explicit type is present, no conversion will be done and an error occurs. The explicit conversion is the fastest method because no method searching is required but the readability of the scripting code is reduced.

5.8 Class Reflection

The information gathered during the parsing process can be inquired at runtime. Each class or object can be analyzed regarding:

- parent and child classes,
- constructors with complete signature,
- methods including complete signature,
- enumerations,
- objects currently instantiated, and
- object relationships.

This information can be used to build sophisticated integrated development environments, including syntax highlighting for classes, methods and enumerations, class hierarchy browser, and run-time object browser (Figure 5).

6 Examples

iVRS represents a complete 3D graphics package for Tcl/Tk. In the following, we illustrate this along several examples.

6.1 3D Object Viewer

As a first example, let us develop a 3D object viewer. It can be used to view and inspect 3D objects constructed with AutoDesk's 3ds max™.

The script below completely implements the 3D object viewer. A snapshot of the application is shown in Figure 5.

```
# loading iVRS extension
package require VRS

# create the root scene node
set myScene [new SceneThing]

# create camera and append to scene
set myCamera [new Camera {0 -2 -2} {0 0 0} 60]
$myScene append $myCamera

# insert light source
set distantlight [new DistantLight]
$myScene append $distantlight

# read 3DS file and put it directly into scene
set my3ds [ObjectLoader readFile dragon.3ds]
$myScene append $my3ds

# create Tk OpenGL canvas
set myCanvas [new TclCanvas .view 400 400]
pack .view

# append root scene node to canvas
$myCanvas append $myScene

# append trackball behavior node to canvas
$myCanvas append [new TrackBall $my3ds]
```

The first command loads the VRS package and initializes VRS classes and wrapper tables. The application is implemented by a scene graph, a behavior graph, and a 3D canvas.

First, we create the root node of the scene graph and store its name in the Tcl variable `myScene`. The scene graph contains a virtual camera, a light source, and the subgraph that represent the 3DS object components. The `Camera` object defines camera position, camera focus and field of view angle. We activate the camera by appending it to `myScene`. To illuminate the scene, we insert a distant light into the scene graph.

VRS provides an object loading mechanism that supports several file formats (e.g., JPEG, TIF, 3D StudioMAX). A call to the static `ObjectLoader` method `readFile`, tries to find a reader for the given file and, if found, returns the object the reader creates. The name of the 3DS data object read from `dragon.3ds` is stored in the variable `my3ds` and inserted into the scene graph. The 3DS data object consists of a node whose node content objects represents geometry and graphics attributes of the 3DS object.

Next, we create an OpenGL canvas to display the scene graph. The `TclCanvas` can be treated as a usual Tk GUI component; it can be integrated in any Tk top-level or container widget. The constructor requires a

well-defined Tk pathname (`.view`). Then, we pack the widget to make it visible.

We link the canvas to the scene graph and to a behavior graph that consists just of a track ball node, which allows users to interactively rotate the 3DS object by mouse motion.

6.2 3D Object Viewer with Shadows

Let us extend the example of the previous section by adding shadows (Figure 6). Shadow rendering is based on shadow maps, a texture-based approach that is now supported by graphics hardware.

```
# loading iVRS extension
package require VRS

# create the root scene node
set myScene [new SceneThing]

# create camera and append to scene
set myCamera [new Camera {0 -2 -2} {0 0 0} 60]
$myScene append $myCamera

# insert light source
set distantlight [new DistantLight]
$myScene append $distantlight

# specify light source that casts shadow
set cast [new ShadowCaster $distantlight]
$myScene append $cast
$myScene append \
    [new ShadowCasterSwitch $cast true]

# read 3DS file and put it directly into scene
set my3ds [ObjectLoader readFile dragon.3ds]
$myScene append $my3ds

# specify objects that receive shadows
set shadowed [new Shadowed $distantlight]
$myScene append $shadowed
$myScene append \
    [new ShadowedSwitch $shadowed true]

# add box to make shadow visible
$myScene append \
    [new Box {-2 -1.1 -2} {2 -1 2}]

# create Tk OpenGL canvas
set myCanvas [new TclCanvas .view 400 400]
pack .view

# append root scene node to canvas
$myCanvas append $myScene

# append trackball behavior node to canvas
$myCanvas append [new TrackBall $my3ds]
```

In VRS, several attribute classes control the shadow rendering technique. In the example, we specify in which scene subgraphs shapes and light sources cast shadows (`ShadowCaster`) and in which scene graphs shapes receive shadows (`Shadowed`). Because shadowing is a global illumination phenomenon, we can control locally shadowing by switch objects (`ShadowCasterSwitch` and `ShadowedSwitch`)

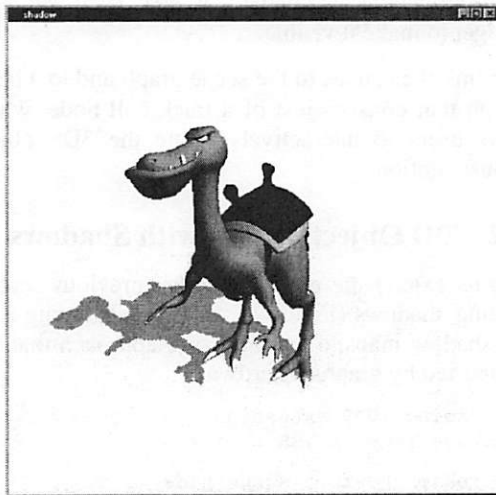


Figure 6: Viewer for 3D objects with shadows.

6.3 Integrating Tcl Scripts as Callbacks

In VRS, scene graphs and behavior graphs can also include nodes that define callbacks for certain types of events (e.g., time events, redraw events). Callbacks are able to call, for example, C functions or object methods. We extended VRS by a specialized callback class that encapsulates a Tcl script and invokes that script if the callback is activated. As a consequence, we can insert Tcl scripts at any position in scene graphs and behavior graphs. This way, the traversal of a graph is not entirely under control of the C++ engines but can be partially defined by scripts.

The following example extends example 6.1 by creating a Tcl callback that saves the current contents of the 3D canvas and writes the contents to an image file. The resulting image files could be compressed to an AVI or MPEG stream.

```
# snapshot proc
proc writeSnapshot {} {
    # make canvas available
    global myCanvas

    # save canvas content to Image object
    set myImage [$myCanvas snapshot]

    # make unique filename
    set filename snap[clock ticks].ppm

    # call static method to write ppm image
    PPMWriter writeFile $myImage $filename
}

# create TclCallback for writeContent
set myCall [new TclCallback writeSnapshot]

# create callback node for scene graph
set myRedraw [new SceneCallback $myCall]

# add SceneCallback to scene graph
$myCanvas append $myRedraw
```

The Tcl procedure `writeSnapshot` is responsible for capturing the current canvas contents into an `Image` object. The snapshot is saved as PPM file under a unique filename. To write the canvas contents after each redraw, a `TclCallback` object is used; it is inserted into the scene graph node `myRedraw` that invokes the callback in the case of a redraw event. Finally, we have to append that new node into the scene graph.

6.4 Development Environments

The API information, which is gathered during API analysis and stored as part of the API mapping, facilitates the construction of development environments. As core parts, we can take advantage of the collected information to build automatically control widgets for objects. The following example shows widgets that query constructor arguments and types, and instantiate GUI components based on this information, which are used to manipulate the initial values of the constructor arguments (Figure 7).

```
# specify the desired class
set what Sphere

# iterate over all constructors
foreach {name types args defs} \
    [VRS info ctors $what] {

    # iterate over all constructor arguments
    foreach t $types a $args d $defs {

        # build GUI components for arguments
        label .l$a -text "$a ($t)" -width 10

        # switch depending on type
        if {[string equal $t double]} {
            scale .e$a -variable $a
        } else {
            entry .e$a -textvariable $a
        }

        pack .l$a .e$a -side top

        # append to argument string
        append ctor_args "\$a "
    }

    # button to create and insert object
    button .b -text Create -command " \
        set obj \[new $what $ctor_args\]; \
        \myScene append \$obj"
    pack .b -side top
}
```

The `VRS info ctors` command returns a list of all constructors of the specified class containing all argument types, argument names, and argument default values. The widget snapshots (Figure 7) show widgets for torus objects (defined by outer radius, inner radius, center, and three aperture angles), sphere objects (defined by radius, cutting planes in y, and aperture angle), box objects (defined by two corner points), and cone objects (defined by height, radius, and aperture angle).

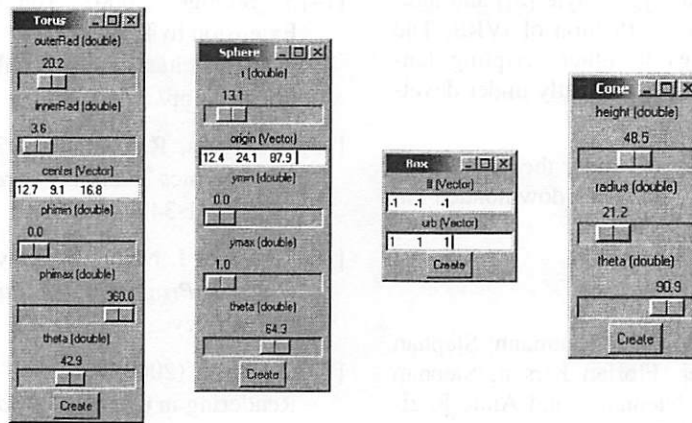


Figure 7: Automatically generated construction widgets for the classes Torus, Sphere, Box, and Cone.

Our mapping technique preserves not only the argument types of methods but also the argument names and their default values. This information is important for documenting the API and provides valuable information when developers want to interactively explore classes, objects and their methods. API information and run-time type information thus allow us to design integrated development environments in a straightforward way. Figure 5 illustrates such an environment built for VRS: Developers can browse classes, edit scripts, explore classes, inspect objects, and run applications.

6.5 An interactive 3D Map System

The mapping technique has been used to bind the APIs of VRS and LandExplorer, a 3D-map library built on top of VRS, to Tcl. Using iVRS and Tcl, we have implemented a complete interactive 3D-map system.

The 3D-map system supports real-time, multi-resolution terrain rendering, multi-texturing of the terrain surface, and integration of 3D objects into the terrain model. In addition, various exploration functions have been added such as information lenses, fly-throughs, and meta-views.

Although real-time terrain rendering is time critical, the API mapping has no noticeable impact on performance because scene graph traversal, the most critical part of the display, is performed within C++. Even if some callbacks of a scene graph or behavior graph use Tcl scripts, the overall performance is not being affected.

As main advantages for building complex 3D graphics applications we observed:

- Easier class, object, and method selection.
- Easier construction of scene graphs and behavior graphs.
- Easier implementation of variants.

- Easier execution of experiments.
- Rapid prototyping.

7 Conclusions and Future Work

The described API mapping technique copes with important C++ language features such as classes, overloaded methods, operator methods, enumerations, and inheritance relations. It also analyzes the API with respect to argument names and default values. A semi-automatic memory management facilitates the usage of objects through the scripting language. Objects used “by value” are treated differently by inline conversion. Furthermore, the mapping technique uses standard Tcl without the need for any object-oriented Tcl extension.

The mapping technique gathers API information (e.g., classes, methods, and arguments) and provides run-time type information (instantiated objects, list of available classes, etc.). Both are the prerequisites for interactive development environments.

As a proof-of-concept, the complete C++ API of the Virtual Rendering System has been mapped to Tcl successfully. Since scripts mainly construct and modify models (e.g., scene graphs, behavior graphs, and node components) but are not directly involved in evaluating these models (e.g., scene graph traversal), applications written with Tcl are almost as efficient as applications written in C++. Even real-time 3D computer graphics such as 3D terrain viewers can be implemented based on the mapped API.

Finally, we observed that developers can use the mapped API more easily than the C++ API because no detailed knowledge of C++ is required and the library’s functionality can be interactively explored and immediately applied.

In our future work, we are going to support additional C++ language features such as namespaces and nested classes. Furthermore, C++ comments of public methods

should be recognized (e.g., doxygen style [6]) and integrated into the run-time class reflection of iVRS. The implementation of mappings to other scripting languages such as Perl or Python is currently under development.

VRS and iVRS are free software under the GNU General Public License and can be downloaded at www.vrs3d.org.

Acknowledgements

We would like to thank Konstantin Baumann, Stephan Brumme, Christian Günther, Florian Kirsch, Stephan Kirsch, Haik Lorenz, Marc Nienhaus, and Anne Roziat for their contributions to VRS.

References

- [1] D. Beazley (1997): SWIG Reference Manual. Department of Computer Science, University of Utah, www.swig.org/Doc1.1/PDF/Reference.pdf
- [2] J. Döllner, K. Hinrichs (1997): Object-Oriented 3D Modeling, Animation and Interaction. *The Journal of Visualization and Computer Animation*, 8(1):33-64
- [3] J. Döllner, K. Hinrichs (2001): A Generic 3D Rendering System. *IEEE Transactions on Visualization and Computer Graphics*, 8(2)
- [4] C. Esperanca, S. Fels, J. Joyner (2000): TkOGL 3.0. www.ece.ubc.ca/~hct/projects/tkog1.html
- [5] W. Heidrich, P. Slusallek, H.-P. Seidel (1994): Using C++ class libraries from an interpreted language. *Proceedings of TOOLS USA '94*, 397-408
- [6] D. van Heesch (2002): www.doxygen.org
- [7] Khronos Group (2001): OpenML Specification. www.khronos.org/OpenML_1-0_Final_Spec.pdf
- [8] M. McLennan (1993): [incr Tcl] - Object Oriented Programming in TCL. *Proceedings of Tcl/Tk Workshop 1993*, Berkeley
- [9] Nebula Game Engine (1998): www.radonlabs.de
- [10] J. Ousterhout (1994): Tcl/Tk, Addison-Wesley
- [11] J. Ousterhout (1998): Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3), 23-30
- [12] F. Pilhofer (1997): Using C++ objects with Tcl. www.fpx.de/fp/Software/tclobj/tclobj-1.2.tar.gz
- [13] W. Schroeder, K. Martin, B. Lorensen (1997): The Visualization Toolkit - An Object-Oriented Approach To 3D Graphics, Prentice Hall
- [14] S. Sinnige (2000): Tclpp: An Object-Oriented Extension to Tcl www.geocities.com/SiliconValley/Network/2836/projects/tclpp/
- [15] P. Strauss, R. Carey (1992): An Object-Oriented 3D Graphics Toolkit. *Proceedings SIGGRAPH '92*, 26(2):341-349
- [16] M. Woo, J. Neider, T. Davis, D. Shreiner (1999): *OpenGL Programming Guide - 3rd edition*, Addison-Wesley
- [17] C. Wynn (2002): Using P-Buffers for Off-Screen Rendering in OpenGL. *Nvidia Technical Paper*

The AGFL Grammar Work Lab

Cornelis H.A. Koster & Erik Verbruggen
Dept. Comp. Sci.
University of Nijmegen (KUN)
The Netherlands
{kees,erikv}@cs.kun.nl

Abstract

The AGFL Grammar Work Lab is the first parser generator for natural languages to be brought under the GNU public license. Apart from its linguistic uses, it is intended for the production of parsers which are to be embedded in application systems. In particular, the AGFL system comes with a free grammar and lexicon of English, allowing the construction of user interfaces and applications involving Natural Language Processing.

We give a brief description of the AGFL formalism and its use in transducing English text to Head/Modifier frames and discuss some possible applications.

1 Introduction

A large part of the capacity of computers is devoted to the capture, storage, analysis, transformation and production of human-readable documents, in the form of publications, correspondence and web-documents. Therefore, a growing number of applications is dependent on, or could benefit from, some form of linguistic analysis of documents.

In particular, Natural Language Processing (NLP) is an important enabling technology for future web-based applications: from classification of web-pages, filtering and narrowcasting to more intelligent search machines and services based on the automatic interpretation of the contents of documents. As is the case in Information Retrieval (IR) in general, the state-of-the-art in search machines on the web is based mainly on the use of keywords, and only some limited linguistic techniques are used to

enhance recall: stop lists, stemming, some ontologies, and the simplest of phrase recognition techniques. An example is the Linguistix software library, incorporated in commercial search machines like Altavista and Askjeeves, which performs tagging, lemmatization and fuzzy semantic matching.

No use is made of syntax analysis or semantic analysis and the discourse structure of documents is largely ignored, although their use might yield an important increase in precision. The great success of the present statistical techniques combined with such “shallow linguistic techniques” [Sparck Jones, 1998] has led to the idea that deep linguistics is not worth the trouble.

What is worse, it is very hard to find resources to build applications using deeper linguistic techniques like parsing. In applied linguistic communities like the corpora list, many groups appear to be in need of parsers and lexica for natural languages, and requests for freely accessible linguistic resources are frequently posed. But such resources are just not available, or just not free.

There is a definite need for parsers and lexica in the public domain, so that people developing say a question answering system do not have to start by reinventing the wheel. The extraordinary success of one such resource, the [Princeton WordNet], may be due to its public availability rather than to superb quality, but it has had tremendous impact, and it is improving over time.

In this article, we make a plea for linguistic resources in the public domain and announce the public availability of the AGFL Grammar Work Lab, and the EP4IR grammar and lexicon of English. We describe how to use a parser, generated by the AGFL system from EP4IR, in practical applications.

2 The problem area

Academic research groups that have developed parsers and lexica are unable to sell these as products and market them. Such resources may have cost many many years for their development, but that does not mean anyone is willing to pay the same price to obtain them. Furthermore, being academics they are not in a position to offer maintenance.

There are a number of repositories of linguistic resources, but they are either proprietary or they make the resources available at a low price for research purposes only, while the conditions for commercial use are very vague (write us). The low price hardly covers the cost of distribution and certainly is not enough to cover maintenance. The gold wagon is expected to come in from industry.

As a consequence, in building NLP applications many industrial corporations prefer to develop their own resources from scratch rather than being dependent on others. Research whose results might become economically interesting can not be based on such resources.

In fact, the situation is remarkably like that for software in the eighties, and the same solution should be considered:

Basic linguistic resources like grammars, lexica, parsers, corpora and ontologies should be made freely available in the public domain, especially if they have been developed with public money. Their users should be invited to contribute improvements, thus enabling a low-cost form of maintenance.

Where have we heard this before?

3 AGFL under GPL

The purpose of this article is to announce the availability of the AGFL Grammar Work Lab under the GNU Public Licence, making it publicly and freely available as a tool for linguistic research and for the development of NLP-based applications. The AGFL system is the *first parser-generator for natural languages* available under the GPL.

The run-time system for the generated parsers has been brought under the Lesser GPL, so that parsers by the system may be included in other systems (even commercially) under very liberal conditions.

The system comes with a number of grammars and lexica for free, in particular the EP4IR (English Phrases for Information Retrieval) grammar of English. Linguists and Computer Scientists alike are invited to use the AGFL system and the accompanying EP4IR grammar and lexicon of English for whatever purpose they like, including commercial purposes, as long as the GPL is adhered to. Linguists are invited to make and share improvements to the free grammars and lexica, or add new grammars and lexica in the same spirit.

4 Affix Grammars over a Finite Lattice

The AGFL formalism (Affix Grammars over a Finite Lattice) [Koster, 1992] is a notation for Context-Free grammars enriched with finite set-valued features, acceptable to linguists of many different schools. For a computer scientist this means: syntax rules are procedures with parameters and a nondeterministic execution, like that of PROLOG.

No natural language can reasonably be described by a deterministic grammar, so that deterministic parser generators like YACC are useless for realistic NLP. Nondeterminism (ambiguity!) is an essential property of language, so that a completely different kind of parser generator is needed. The AGFL system is such a system.

For the interested reader we give two examples to convey some of the flavor of AGFL. The notation of AGFL is reminiscent of that of PROLOG, with which it is distantly related. This may help in reading (and understanding) the examples.

4.1 Example: noun phrases

The first example is a fragment of a rather simplistic grammar for english noun phrases. Each rule is exemplified by one or more examples.

NUMB :: sing; plur.

The feature expressing the number can take on two values: sing or plur.

```
RULE noun phrase (NUMB):  
  noun part (NUMB).  
  # EX the previous president
```

```
RULE noun phrase (plur):  
  noun part (NUMB1), coordinator,  
  noun phrase (NUMB2).  
  # EX the president and his wife
```

These two rules express that a noun phrase consists of one or more noun parts combined by coordinators. In the latter case it is always plural.

```
RULE noun part (NUMB):  
  determiner (NUMB), noun group (NUMB);  
  # EX the red bag  
  noun group (NUMB).  
  # EX software engineering
```

This rule has a number of alternatives, separated by semicolons. The number of the determiner has to agree with that of the noun group.

```
RULE noun group (NUMB):  
  noun (NUMB);  
  # EX bag  
  adjective, noun group (NUMB);  
  # EX red bag  
  noun group (NUMB1), noun (NUMB).  
  # EX software engineering
```

Obviously, the last rule is ambiguous for a noun phrase consisting of three or more nouns, like software engineering conference. Other sources of ambiguity are found in the attachment of preposition phrases (not described here) and in lexical ambiguities (e.g. time as noun and verb). AGFL provides a number of mechanisms (penalties, lexical frequencies, syntactic probabilities) to help in finding the most probable analysis (rather than the set of all analyses).

4.2 Example: transduction

The second example shows the recognition of certain sentence patterns and their *transduction* to Head/Modifier pairs [head, modifier]. The transduction mechanism allows the production of a

(compositional) translation instead of a parse tree. For each alternative of the rule, a transduction can be specified preceded by a slash.

```
NUMB:: sing | plur.  
PERS:: first | secnd | third.  
TRAN:: intrans | trans.
```

```
RULE sentence:  
  subject(NUMB, PERS),  
  verb part (intrans, NUMB, PERS) /  
  "[", subject, ",", verb part, "];
```

The *transitivity* feature of the verb determines whether it takes an object. Assuming a suitable transduction for subject and verb part, this rule would transduce I am freezing to [I, freeze].

```
RULE sentence:  
  subject(NUMB, PERS),  
  verb part (trans, NUMB, PERS), object /  
  "[", subject, ",", verb part, ",", object, "];
```

Under similar assumptions, this should transduce I was attending a software engineering conference to [I, [attend, [conference, software engineering]]].

5 The EP4IR grammar

The “English Phrases for IR” (EP4IR) grammar is a reasonably complete grammar of English, concentrating on the description of the noun phrase and the verb phrase. The grammar is provided with a large lexicon, providing detailed Part-Of-Speech information. The grammar is quite robust against incorrect input and unknown words. The EP4IR grammar and lexicon were developed in the [PEKING project] for Information Retrieval applications, and they are released along with the AGFL system.

From the grammar and lexicon, an English parser can be generated automatically using the AGFL system, which produces as its output not parse trees but Head/Modifier frames, more or less as described above. The following picture illustrates the generation of a parser and its use.

The HM frame representation is a very good starting point for diverse applications, and the transduc-

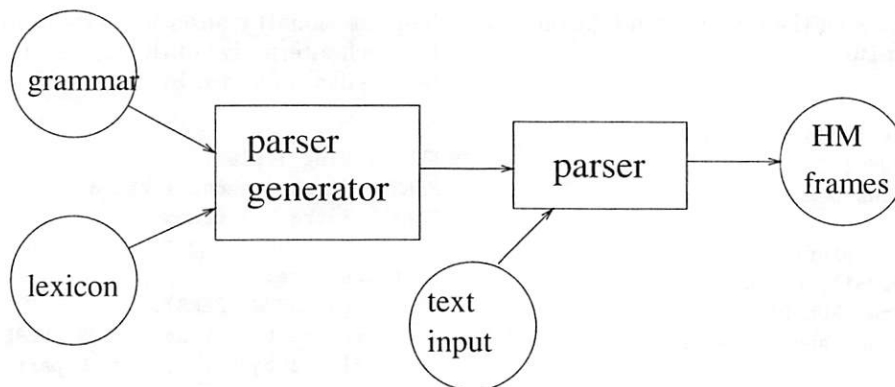


Figure 1: Generating and using a parser

tion can be adapted with relative ease to yet other applications.

The grammar still many details of the language, which will hopefully seduce linguists to propose additions and improvements.

5.1 Head/Modifier frames

The frames generated when parsing some text represent only the major relations expressed in the text:

relation	example
subject	[N:script,V:calls]
relation	[P:it,V:reduced]
object	[V:performed,N:research]
relation	[V:interrupted,P:it]
attr/pred	[N:divergent,N:function]
relation	[N:analysis,A:syntactic]
preposition	[V:interacts,with N:user]
relation	[N:interaction,with N:user]
	[A:relative,to N:counter]

Notice the fact that each word is *typed* as Noun, Pronoun, Adjective or Verbform and that, except for the prepositions, *all other words are eliminated*: adverbs, determiners, quantifiers. What is left is really the bare bone structure of the text. The goal is to get rid of embellishments and variations which add no value for retrieval purposes.

The types may be used in further processing of the frames, or they may be removed by some postpro-

cessing.

A demo version of the parser/transducer is available at the website of the [AGFL project]; trying it out may help in understanding the HM frame representation.

5.2 Nested frames

Due to the recursive nature of phrases, the frames transduced from them may be nested, e.g. (omitting the types):

IBM sponsored this conference

⇒

[IBM,[sponsored,conference]]

this conference was sponsored by IBM

⇒

[IBM,[sponsored,conference]]

Every PhD student gets a reduced price for the ETAP conference

⇒

[[student,PhD],[get,[price,reduced|for
conference,ETAP]]]]

Notice the transformation from passive to active voice in the second example. Nested frames can be unnested using a component included with the grammar, the *unnester*, which for the last example yields

[N:student,N:PhD] [N:student,V:gets]
[V:gets,N:price] [P:it,V:reduced]

```
[V:reduced,N:price] [N:price,  
for N:conference] [N:conference,N:ETAP]
```

5.3 Application examples

We shall describe only a few of the many possible examples, leaving it to the reader to contrive others.

- **Information Retrieval**
The grammar was developed for IR applications, in which the traditional keywords are to be replaced by frames obtained from phrases. All frames are first unnested, so that each document is represented by a bag of frames without nesting. The frames are also morphologically normalized, using a lemmatizer and the typing information provided. Furthermore, semantically related frames will be clustered together (see [Koster et al., 1999]).
- **Information Analysis and Modelling**
An important step in most analysis techniques is to start out from an informal (i.e. verbal) description of the problem domain or the problem, and scan it for nouns and verbs: the nouns are candidate objects or classes and the verbs are candidate methods [Abbott, 1983]. Other elements, like the adjectives can also be used [Graham, 1994]. The HM frame representation obtained by using EP4IR can be used straight away for this purpose. The same technique can be used for the *validation* of an existing Object-Oriented Analysis model [Frederiks et al, 1996].
- **Question answering system**
A question answering system of any kind will need more information than is included in the HM frames, e.g. quantifiers and determiners. Of course the grammar does already recognize these constructs, so by modifying three or four lines in the transduction they will also be expressed. Luckily, the grammar already knows how to parse questions.

Feel free to use the AGFL system for your purposes according to the GPL/LGPL license. Let us know if you have a nice application. For more complicated projects you might consider collaborating with the authors of this paper.

6 Disclaimer

Nobody is perfect. The currently available release 2.0 of the AGFL system, resulting from a total revision of the formalism and its implementation, is only the first step. It still has to be improved, in particular with respect to its speed, but we are working on that. A version generating much faster parsers is in the pipeline. In the mean time, the system is "solidly under way, but may not yet be 100% finished". The same holds for the accompanying grammars and lexica.

7 Summary and conclusions

There are many good reasons to bring the AGFL Grammar Work Lab into the GNU family:

- there is at present no parser generator for linguistic grammars under GPL
- AGFL can fill a niche that will make GNU attractive to a large number of linguistic users who now live in a Microsoft-dominated world
- AGFL is a well-developed and stable system, which merits availability in the public domain
- a university like ours (the University of Nijmegen) is not in a position to distribute and maintain the system on a commercial basis
- The GPL conventions provide a rational framework for its distribution and use
- the open availability of the source text will invite contributions by others, improving the AGFL software and the associated grammars and lexica, which will ease the maintenance problem.

It is our expectation that the availability of a parser generator for natural language parsers in the public domain will enable not only the development of many new applications, but that the good example of making the software system and the associated grammars and lexica freely available will inspire others to contribute grammars and lexica to the public domain. For computer scientists, this argument may be all too familiar, but for linguists this is a wholly new approach!

8 Acknowledgements

The AGFL formalism was first implemented between 1991 and 1996 with funding from the Dutch national research organization NWO. In the period 2000/2001, with financial support from the [NLnet foundation], the formalism has been revised in the light of experience and been brought under the GNU public licence.

Out of the many people who have contributed to the AGFL project we feel particularly obliged to Arend van Zwol, Arjan Knijff and Caspar Derksen who have spent years of their life on this elusive ideal parser generator.

9 Availability

Further information, the EP4IR grammar and all the software can be found at the [AGFL project] website.

References

- [Abbott, 1983] R. J. Abbott: *Program design by informal English descriptions*; CACM 26(11), pg 882-94.
- [AGFL project] <http://www.cs.kun.nl/agfl/>.
- [Frederiks et al, 1996] P.J.M. Frederiks, C.H.A. Koster, and Th.P. van der Weide. Validation of Object-Oriented Analysis Models using Informal Language. Technical Report CSIR9609, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, May 1996. <http://citeseer.nj.nec.com/frederiks96validation.html>
- [Graham, 1994] Ian Graham, *Object Oriented Methods*, AddisonWesley, 1994
- [Koster, 1992] Cornelis H.A. Koster (1992), Affix Grammars for Natural Languages. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Springer LNCS*, pp. 469-484.
- [Koster et al., 1999] C.H.A. Koster, C. Derksen, D. van de Ende and J. Potjer, Normalization and matching in the DORO system. Proceedings BCS-IRSG 1999 colloquium, Glasgow University.
- [NLnet foundation] <http://www.nlnet.nl/>
- [Sparck Jones, 1998] K. Sparck Jones (1998), Information retrieval: how far will *really* simple methods take you? in: Proceedings TWTL 14, Twente University, the Netherlands, pp. 71-78.
- [PEKING project] see <http://www.cs.kun.nl/peking>.
- [Princeton WordNet] <http://www.cogsci.princeton.edu/wn/>

SWILL: A Simple Embedded Web Server Library

Sotiria Lampoudi and David M. Beazley

Department of Computer Science

University of Chicago

Chicago, Illinois 60637

`{slampoud,beazley}@cs.uchicago.edu`

Abstract

We present SWILL, a lightweight programming library that adds a simple embedded web server capability to C and C++ programs. Using SWILL, it is possible to add Internet accessibility to programs that are poorly matched to more traditional methods of web programming such as CGI scripting or web server plugin modules. SWILL also makes it easy for programmers to add web-based monitoring, diagnostics, and debugging capabilities to software not normally associated with internet programming. We like to think of SWILL as an attempt to turn the problem on its head: traditionally, the web server came first, the “programs” later; in our approach, the application is written first, and the server integrated last. For some types of applications, this approach is far more painless. In this paper, we provide an overview of the SWILL library and describe how we have used it to provide web access to a variety of applications including scientific simulation software, a compiler, and a hardware emulator for teaching operating systems.

1 Introduction

With the growth and popularity of the web, many software developers are interested in building web-based interfaces to their applications. However, much of the effort involved in doing this seems to be spent figuring out how to integrate an application into an existing web server using CGI scripts, server plugin modules, or some sort of middleware layer. Although it is certainly possible to implement a web interface in this manner, a programmer may have to contort their application or implement a complicated runtime environment to make it work. To further complicate matters, there are certain applications in which web access would be useful, but which do not really fit into the standard mold of an

internet application. Examples might include simulators, long running scientific programs, and compilers.

Instead of figuring out how to integrate an application with a web server, an alternative approach might be to flip the whole situation around and to ask what it would take to add a web server to an application as a programming library. If a web server could be added as a library, it might greatly simplify the task of creating web-accessible programs. For instance, since the web server would be part of the application itself, there would be no need to worry about the installation and configuration of a complicated web server environment. A library would also make it easier to add web capabilities to programs that are not really intended to be internet applications, but where a web interface would be useful in providing remote accessibility or as a simple diagnostics tool.

In this paper, we describe SWILL (Simple Web Interface Link Library). SWILL is a lightweight library we have developed that makes it easy to add a web server to C/C++ programs. It consists of a handful of C functions that are inserted into an application to make it web accessible. The implementation is intentionally minimal and is primarily intended for developers who would like to create a web interface without a lot of hassle.

SWILL was initially written for use with high performance scientific simulation software [4]. However, the implementation is generic enough to be used in a variety of other applications—limited only by the programmer’s imagination. For example, we have used SWILL in a compiler project to help with the debugging of parse tree data. It has also been used to monitor the internals of a hardware simulator for teaching operating systems.

The rest of this paper provides a brief overview of SWILL followed by some examples of how we have used the library. At the end, we provide a brief overview of related work and future plans.

2 Library Introduction

The best way to introduce SWILL is with a simple example. Although the library contains about 25 functions, only a handful are needed to get started as shown in this “Hello World” example:

```
#include <swill.h>
void hello(FILE *f) {
    fprintf(f, "Hello World!\n");
}

int main() {
    swill_init(8080);
    swill_handle("hello.txt", hello, 0);
    while (1) {
        swill_serve();
    }
}
```

In this example, the web server is activated by calling `swill_init()` with a TCP port number. One or more documents are then registered with the server using `swill_handle()`. `swill_serve()` is then used to wait for a connection. When a connection is received, an appropriate handler function is invoked depending on the URL. In this case, a request for the document “hello.txt” invokes the `hello()` function and produces the output that you expect.

If an application wants to perform other work in addition to checking for connections, a non-blocking polling function is used. For example:

```
while (1) {
    swill_poll(); /* Requests? */
    ...
    /* other work */
    ...
}
```

This approach allows the web interface to be easily inserted into applications that provide their own event or computation loops. For example, a scientific simulation might call `swill_poll()` at selected intervals during computation, but spend the rest of its time crunching numbers.

There are no restrictions on the type of output a SWILL handler function may produce. However, the document type is implicitly determined by the suffix supplied to the `swill_handle()` function. For example, if a function produces HTML, the following code would be used:

```
void hello(FILE *f) {
```

```
    fprintf(f, "<HTML><BODY>\n");
    fprintf(f, "<b>Hello World!</b>\n");
    fprintf(f, "</BODY></HTML>\n");
}

int main() {
    ...
    swill_handle("hello.html", hello, 0);
    ...
}
```

Similarly, the following code produces a PNG image using the freely available GD library [7]:

```
void
image(FILE *f) {
    int black, white;
    gdImagePtr im;

    im = gdImageCreate(64, 64);
    black = gdImageColorAllocate(im,
                                  0, 0, 0);
    white = gdImageColorAllocate(im,
                                  255, 255, 255);
    gdImageLine(im, 0, 0, 63, 63, wht);
    ...
    gdImagePng(im, f);
    gdImageDestroy(im);
}

int main() {
    ...
    swill_handle("image.png", image, 0);
    ...
}
```

Since a programmer may want to reuse the same handler function for different web pages, SWILL allows an optional pointer to be passed to the handler functions. This pointer is used to pass application specific data or an object to the handler. For example, an application that created various types of data plots might look roughly like this:

```
void
make_plot(FILE *f, void *clientdata) {
    Plot *p = (Plot *) clientdata;
    ...
    // Generate plot
    ...
    write_plot(p, f);
}

int main() {
    ...
    e = new energy_plot();
}
```

```
d = new density_plot();
...
swill_handle("energy.png",make_plot,e);
swill_handle("density.png",make_plot,d);
...
```

Although SWILL is primarily intended for dynamic content generation, it can also deliver individual files or files from a user-specified directory. For example, if a programmer wanted to register a specific file with the server, they would do this:

```
swill_file("foo.html","./htdocs/foo.html");
```

Similarly, a directory of files is registered as follows:

```
swill_directory("./htdocs")
```

When a directory is registered, SWILL delivers files much like a traditional web-server.

In certain applications, a programmer might want to receive HTTP query variables as input parameters (as might be supplied from an HTML form). SWILL automatically parses HTTP query strings in both GET and POST requests. To access query variables as strings, the following function is used:

```
char *swill_getvar(const char *name);
```

However, a more convenient way to get form variables is to use `swill_getargs()` as shown in this example:

```
void adder(FILE *f) {
    double x,y;
    if (!swill_getargs("d(x)d(y)",&x,&y)) {
        fprintf(f,"Missing values!\n");
        return;
    }
    fprintf(f,"%g + %g = %g\n", x,y,x+y);
}
```

The argument to `swill_getargs()` is a format string that specifies the types and names of form variables to be converted. If available, the variables are decoded, placed into C variables, and a success code returned.

3 Concurrency and I/O

SWILL is a single-threaded server that does not rely upon concurrency mechanisms such as forking or multithreading. This limitation is by design and is related to the goal of having a server that could

be easily embedded into a variety of specialized applications such as parallel scientific codes, hardware emulators, and so forth. In these situations, the use of concurrency can introduce serious reliability and portability problems. For instance, an application may not be thread-safe, making it impossible to reliably execute a handler function in parallel with normal execution. Similarly, forking may be impractical in applications with heavy resource utilization or which rely upon interprocess communication (e.g., message passing).

Because of the single-threaded execution model, the implementation of SWILL relies entirely upon non-blocking I/O with timeouts. This prevents the server from indefinitely blocking the application in the event of bad connections and missing data. For instance, if a connection is made, but no HTTP headers are received, SWILL automatically closes the connection after a timeout and returns. The timeout is fully configurable by the user and can be set to only a few seconds if desired.

The I/O for handler functions relies upon a temporary file created with `tmpfile()`. When requests are serviced by handler functions, output is placed into this file. When a handler function has finished execution, the file contents along with HTTP headers are sent back to the client. Normally, SWILL simply passes a corresponding `FILE *` object to handler functions so that they can perform I/O. As an optional feature, SWILL can also capture standard output. This is enabled in `swill_handle()` by prefixing the document name with `stdout:` as follows:

```
swill_handle("stdout:foo.html",foo,0);
```

In this case, all I/O operations on `stdout` are captured for the duration of the handler function. This capture is sufficiently powerful to allow other programs to be executed using `system()` and to have the output of those programs redirected to a web page. For example, the following handler function would capture the output of a system command:

```
void listfiles() {
    system("ls -l");
}
...
swill_handle("stdout:files.txt",listfiles,0);
```

4 Security and Reliability

SWILL is not appropriate for use in applications that require a high degree of security since no support for SSL or cryptographically secure user authentication is provided. However, the library does

provide a few simple security features to restrict access. First, basic HTTP user authentication is available by registering names and passwords like this:

```
swill_user("dave","iluvschlitz")
```

Second, IP filters can be used to disallow or allow connections from specific IPs or ranges of IPs. For example:

```
swill_allow("127.0.0.0");
swill_deny("128.135.11.");
swill_deny("128.135.11.8");
swill_deny("");
```

SWILL also allows users to register a log file in which requests will be recorded and which can be monitored to check for suspicious activity.

Due to the lack of concurrency, a SWILL application may be vulnerable to a denial of service attack. However the library does take reasonable precautions to allow an application to make progress. As mentioned in the previous section, all I/O operations involve non-blocking system calls with timeouts. Therefore, it is not possible for a client to indefinitely block execution by keeping the connection open without transmitting any data. SWILL is also quick to close connections if it detects malformed data such as bad HTTP headers or garbled input. We have considered the possibility of automatically blocking IP addresses that repeatedly send bad requests. However, this is not implemented at this time. Given that IP filters can be used to block access, a user already has the means to restrict access to a set of known hosts.

Finally, since SWILL is embedded in an application, it is certainly possible for bad programming to break the server. For instance, a poorly written handler function could enter an infinite loop or start a computation that exceeds available machine resources. In this case, the application would become unresponsive and would probably die. SWILL does not take any steps to prevent such problems. However, these can be anticipated with a certain amount of common sense, error checking, and having an understanding of the underlying execution model of the application.

5 Parallelized SWILL

A very useful feature of SWILL is that it supports SPMD-style parallel applications that utilize MPI (MPICH is currently supported) [8]. This allows

it to be used on Beowulf clusters and large parallel machines. If used in this style, every node calls `swill_poll()` in parallel which results in a global synchronization. If an incoming request is received, it is forwarded to all of the nodes which then execute the handler function in parallel.

Using SWILL in this setting is no more difficult than in the single processor setting; the HTTP client connects to the master node of the computation, issues a request and receives sorted output collected from all nodes. Under the hood the implementation is also quite simple. The master node (`MPI_Rank == 0`) receives requests and broadcasts them to all of the other nodes. Each node runs the handler function in parallel after which the content is collected by the master node and served in a coherent manner to the HTTP client.

In parallel scientific programming performance is the foremost consideration. To evaluate the performance of `swill_poll()` we distinguish three cases: a) the case in which there is no pending HTTP request, b) the case where there is a pending request for a file or authentication, and c) the case where there is a request for dynamic content.

- a) This case is quite simple. If there is no pending HTTP request, the master node communicates a null request to all nodes, and the host code continues execution immediately after the call to `swill_poll()`. There is one synchronization in this case, for the call to broadcast the null request.
- b) This case is also handled with an overhead of one synchronization. The master node identifies the pending request as one that can be served by it alone, serves it and broadcasts a null request. This also reveals one of our underlying assumptions, namely that the SPMD program is running on a shared filesystem of some sort, so that it would make no sense to return n (for n nodes) copies of the requested file. This behavior is easy to get around. If it is desirable that a copy of the requested file be returned from *each* node, the user can just write a function that reads the file in and prints it to stdout or uses `swill_printf()`.
- c) This is a somewhat more expensive operation. The master node broadcasts the pending request to all nodes, who parse it and execute the appropriate function. Then each node transmits the result of its execution to the master. Next, the master node serves the HTTP response and resumes execution of the host code,

while the back-end nodes resume computation immediately. All in all, this case has a cost of one broadcast plus n (for n nodes) communications.

In our quest for simplicity we have left it up to the user to produce output that denotes what process each segment of the output is coming from – SWILL merely orders it in rank order and serves it.

One interesting use of the parallel capability of SWILL is in overcoming limitations – whether due to architecture or policy – imposed by the administrators of clusters and supercomputing centers. Often it is not possible to access the backend computational nodes of a supercomputer or cluster in order to query the state of one's execution – at least not through a normal shell. When an application embeds a web server, all that is required is knowledge of the master node and access to an unprivileged port. Web requests can then be easily translated into operations that execute on each node of the system.

6 Applications

SWILL has primarily been developed as a tool for remote process monitoring, debugging, and diagnostics. This section describes some applications in which we are using the library.

6.1 Scientific simulation monitoring

The motivating application for most of SWILL's development has been that of monitoring long-running scientific simulations. These programs are typically non-interactive batch jobs that provide little in the way of user feedback. However, to make sure a program is running correctly, a scientist may want to periodically monitor the state of their programs. For example, they might want to check for numerical instabilities or to see how far an experiment has progressed.

To do this, a simulation can be modified slightly by implementing a few handler functions and inserting `swill_poll()` calls into selected places in the simulation loop. For example:

```
/* Initialize SWILL */
swill_init(3737);
/* Register a few handlers */
swill_handle(...);
swill_handle(...);
...
```

```
for (i = 0; i < nsteps; i++) {
    compute_forces();
    integrate();
    boundary_conditions();
    redistribute_data();
    if (!(i % output_freq)) {
        write_output();
    }
    /* Check for connections */
    swill_poll();
}
```

Although this is only a simple example, this technique is easily extended to provide a variety of advanced monitoring capabilities. For instance, if a graphics library is available, the web interface can be used to generate on-the-fly plotting and data visualization. Web access can also be provided to temporary files and other debugging output as the simulation runs. Using HTTP query variables and forms, a scientist could even alter various simulation parameters, enable diagnostic features, temporarily suspend computation, and so forth.

6.2 Compiler parse tree browsing

As a more unusual example, we have used SWILL to provide a web interface to SWIG, a compiler for creating scripting language extensions [3]. One of the challenges of compiler implementation is that of creating, traversing, and managing parse tree data. For the purposes of debugging, it is fairly common to dump the parse tree into a text file where it can be examined. Unfortunately, even for small input files, this might generate a lot of output since a parse tree might contain hundreds to thousands of nodes. This makes it difficult to find the specific information of interest.

As an alternative to dumping the parse tree to a file, a more convenient way to examine the parse tree data is to run SWIG using a special `-browse` option like this:

```
$ swig -browse -c++ -python example.i
SWIG: Tree browser listening on port 4908
```

In this mode, the compiler enters a web-server mode after all parsing and code generation stages have been completed. Then, by pointing a browser at the appropriate port number, it is possible to point-and-click through internal parse tree data. A sample screenshot of this interface is shown in Figure 1. Unlike the information in a text dump, the web interface provides a more more detailed picture of how

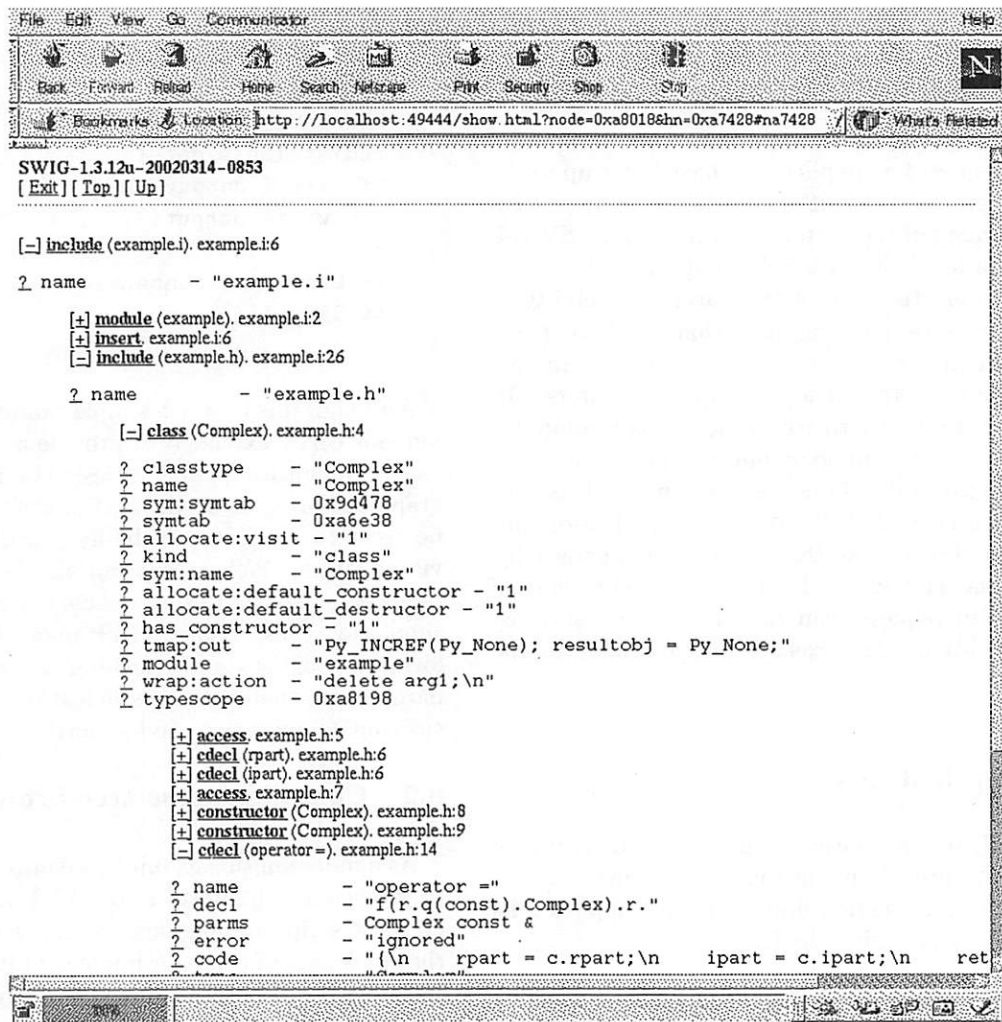


Figure 1: Parse-tree browsing in SWIG

information is organized in the compiler. For instances, pointers are represented by hyperlinks and clicking on a link is essentially the same as dereferencing a pointer in C. The web interface also provides access to compiler symbol tables, type tables, and other information.

At first glance, the idea of a web-enabled compiler sounds crazy. However, this capability greatly simplifies debugging and development of the compiler. Furthermore, a web browser interface works perfectly well as a simple data exploration tool and it was extremely easy to implement—requiring only a half day of effort and a few hundred lines of code. In comparison, the development of a customized tree browser using a GUI toolkit such as Tcl/Tk would have been a much more involved project, would have greatly complicated the configuration of the compiler, and would have provided little if any extra functionality.

6.3 Operating systems project

At the University of Chicago, the course in operating systems requires students to implement a simple Unix-based kernel that runs within an emulated hardware environment. For emulation, we use a modified version of Yalnx, an emulator originally developed by Dave Johnson at Rice University [11]. In this project, students are given eight weeks to implement a kernel from scratch including boot loading, virtual memory management, I/O device drivers, processes, and interprocess communication. Kernels are implemented in C and typically consist of a few thousand lines of code.

One of the problems of working with the emulator is that debugging is extremely difficult. For the most part, the only available diagnostics are an optional hardware trace file and the output of print statements included in the student's imple-

mentation. Unfortunately, this information is often incomplete—making it very difficult to reconstruct the system state that might be the source of a problem.

As an experiment, we have recently used SWILL to add a web-interface to the Yalrix emulator. Internally, Yalrix relies heavily on advanced features of signal handling on Solaris. Specifically, user-mode programs execute natively on the SPARC whereas the student kernel runs in response to signals such as SIGSEGV and SIGALRM (which are transformed into “hardware interrupts”). To instrument the emulator with a web server, we implemented a number of handler functions, initialized the server on startup, and placed a `swill_poll()` call into the SIGALRM handler that is used for internal timing of the emulator.

Using the web interface, it is possible to directly connect to the emulated hardware. Available information includes the current status of all hardware registers, page table settings, and the contents of physical memory pages. It is also possible to pause execution and to obtain traces of recent hardware operations (the history of memory mapped I/O ports, registers, and interrupts). More importantly, the web interface allows you to observe system behavior that is nearly impossible to obtain otherwise. For instance, by watching page tables you can easily spot kernel memory leaks and other inefficiencies in the implementation.

7 Discussion of Related Work

Internet programming is obviously a huge topic, making a detailed comparison of SWILL to related work difficult. Overall, we feel that SWILL differs from other work in a number of respects. First, a considerable amount of attention has been given to programming techniques such as CGI scripting, web server modules, server pages, and programming environments for building Internet applications [10, 2, 6, 12, 17]. Although these techniques are successfully used to incorporate programming libraries and other applications into a larger Internet framework, SWILL has a somewhat different focus than this. Instead of trying to build Internet applications, SWILL is mostly concerned with providing Internet access. This may be a subtle point, but the applications for which SWILL is the most useful are not really designed for the Internet—Internet access is merely an add-on feature that can enhance them.

SWILL might also be compared to work in distributed computing. For instance, SOAP and XML-

RPC are often mentioned as mechanisms for adding internet access to applications [13, 16]. In fact, toolkits such as gSOAP can be used to simplify the integration of an application into such an environment [14]. The problem with these approaches is that they are mostly focused on the problem of turning an application into some sort of pluggable network service to be used within a complicated middleware layer. SWILL, on the other hand, is much more lightweight and is oriented more towards end-users. For instance, users merely connect to the server and are presented with application-specific information in a format that is easy to use and manipulate. There is no hidden application framework or network layer at work.

Finally, SWILL is closely related to domain-specific efforts in providing remote access to applications. For instance, in scientific computing, a lot of attention has been given to the area of “computational steering” [15]. One of the primary goals of steering research is to provide fine-grained interactivity and user feedback to scientific software that is normally batched-oriented and non-interactive. Traditionally, this work has relied upon customized network protocols, complicated client software, and high-end hardware such as graphics workstations. (For a detailed discussion of computational steering we refer the reader to the excellent article [9].)

In some cases, application frameworks may provide web access for this purpose. The Cactus code [5, 1], a modular scientific programming framework, is such an example (the CactusConnect/HTTP and CactusConnect/HTTPExtra “thorns” – modules – are supposed to provide HTTP access to a cactus application). In such cases, however, the web access features are not usable as a stand-alone library. If one wants to have web access, one is forced to buy into a framework with all the pain and risks that entails. SWILL tries to avoid this by focusing exclusively on the problem of web access.

8 Implementation Details

SWILL is implemented entirely in ANSI C and consists of about 2500 semicolons. Most of the implementation (1500 semicolons) simply provides a small set of generic data structures (hashes, lists, strings, etc.) that are used elsewhere. The library itself requires minimal memory overhead. However, all of the generated web pages involve internal buffering. Therefore, memory use is directly proportional to the size of generated web pages. Clearly the library would be inappropriate for serving huge

amounts of data. However, this was not a design goal.

The performance overhead of using SWILL depends on frequency of polling (and obviously the number of incoming connections). On a single CPU, `swill_poll()` is nothing more than a thin wrapper around the `select()` system call. With MPI, polling requires a barrier synchronization across processors. This is obviously more expensive and careful consideration must be given to parallel applications.

For networking, SWILL relies upon the HTTP/1.0 protocol. Although this is less powerful than HTTP/1.1, it is easier to implement and perfectly well suited for most situations.

9 Limitations

SWILL was primarily designed as a tool for building quick-and-easy web interfaces to C/C++ programs. Its single threaded execution would be inappropriate for a high-traffic web site and you probably would not want to use it as the basis of a large internet application. Similarly, internal buffering and other aspects of the implementation make the server inappropriate for delivering very large amounts of data. The server is also unable to support data streaming or any sort of application in which the HTTP connection would be kept alive over a prolonged period.

Although SWILL provides some basic security mechanisms, it would not be appropriate for applications in which security was critical. It should also be added that firewalls and other security mechanisms may prevent users from accessing a server if access to user TCP ports is blocked. Obviously, SWILL cannot address these problems of social engineering.

10 Future Plans

In our own work, SWILL has proven to be remarkably simple and effective to use. Therefore, we have every intention of preserving the minimal nature of the implementation. However, it may be interesting to provide alternative interfaces to C++ and Fortran. Since SWILL does not rely upon anything more than a few simple functions and standard I/O operations, it would be relatively easy to implement handler functions with a slightly different calling convention. For example, in C++,

SWILL might encapsulate I/O in an `iostream` object instead of a `FILE *`.

We have also considered the idea of allowing each SWILL-server to “phone home” to a user-defined master server. If a user was running many different web-enabled applications, this scheme might make it easier to keep track of where they are running. For example, a user could simply connect to the master server and jump to a specific application from there.

Obvious improvements could be made to the underlying HTTP protocol such as support for HTTP/1.1, secure sockets, and digest-based user authentication. However, supporting such features would introduce a lot of extra complexity and would probably offer only marginal benefits in return.

11 Availability

SWILL is freely available under a LGPL licence. More information is available at:

<http://systems.cs.uchicago.edu/swill>

12 Acknowledgments

We would like to thank the reviewers for their helpful comments. Mike Sliczniak and Hasan Baran Kovuk contributed to early parts of the SWILL implementation.

References

- [1] G. Allen et. al., *Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus*, Supercomputing, Denver, CO, (2001).
- [2] Apache Web Server, <http://www.apache.org>.
- [3] D.M. Beazley, *SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++*, 4th Annual Tcl/Tk Workshop, Monterey, CA (1996).
- [4] D.M. Beazley and P.S. Lomdahl, *Controlling the Data Glut in Large-Scale Molecular Dynamics Simulations*, Computers in Physics, Vol. 11, No. 3. (1997), p. 230-238.
- [5] Cactus Code, <http://www.cactuscode.org>.
- [6] C Server Pages, <http://tesitra.com/cserverpages>.

- [7] GD, <http://www.boutell.com/gd/>.
- [8] W. Gropp, E. Lusk, A. Skellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, (1999).
- [9] W. Gu, J. Vetter, K. Schwan. *Computational steering annotated bibliography*, Sigplan notices, 32 (6): 40-4, (1997).
- [10] S. Gundavaram, *CGI Programming*, O'Reilly & Associates Inc., (1996).
- [11] D. B. Johnson, *Yalrix: An Undergraduate Operating System Course Environment and Project Set*, Department of Computer Science, Rice University.
- [12] E. Meijer and D. van Velzen, *Haskell Server Pages: Functional Programming and the Battle for the Middle Tier*, Proc. Haskell Workshop, (2000).
- [13] SOAP, <http://www.w3.org/TR/SOAP/>.
- [14] R.A. van Engelen, K.A. Gallivan, *The gSOAP Toolkit for Web Services and Peer-to-Peer Networks*, Proc. IEEE CC Grid Conf., (2002).
- [15] J. Vetter, K. Schwan, *High Performance Computational Steering of Physical Simulations*, Proc. Int'l Parallel Processing Symp., Geneva, pp. 128-132, (1997).
- [16] XML-RPC, <http://www.xmlrpc.com>.
- [17] Zope, <http://www.zope.org>.

Linux NFS Client Write Performance¹

Chuck Lever, *Network Appliance, Incorporated*
<cel@netapp.com>

Peter Honeyman, *CITI, University of Michigan*
<honey@citi.umich.edu>

Abstract

We introduce a simple sequential write benchmark and use it to improve Linux NFS client write performance. We reduce the latency of the `write()` system call, improve SMP write performance, and reduce kernel CPU processing during sequential writes. Cached write throughput to NFS files improves by more than a factor of three.

1. Introduction

Network-attached storage (NAS) is an easy way to manage large amounts of data. Applications access data stored on NAS via various standard Internet protocols such as HTTP and NFS [6, 20, 21]. To make network-attached storage compete with locally attached storage requires that NAS provide equivalent performance. NAS server performance is well understood, but client performance has long been *terra incognita*.

As Linux servers proliferate within enterprise information infrastructures, performance of the Linux NFS client emerges as a factor critical to the success of complex applications such as database and mail services that use network-attached storage. Efficient access to shared data in laboratories that make extensive use of Linux workstations also depends on superior NFS client performance.

We are interested in two equally important goals. Our first goal is narrow: to improve the performance of the Linux NFS client. To do this, we also pursue a broader goal of identifying factors that influence NFS client performance.

To understand NFS client performance issues, we developed a simple file system benchmark that measures write latency and throughput. In this paper, we describe and rationalize such a benchmark, and use it to identify several means to improve application write performance to files accessed via the Linux NFS client. We also suggest ways to apply the benchmark and comparative techniques to client performance in general.

The remainder of this paper is organized as follows. In Section 2, we detail the development of the benchmark and identify issues that distinguish client from server performance benchmarking. In Section 3, we use this benchmark to expose and correct latencies in the Linux `write()` system call. In Section 4, we outline future areas of exploration and conclude the paper.

2. Measuring NFS client performance

In this section we develop a rationale for a simple sequential write benchmark based on Bonnie [1]. This benchmark was developed on specialized hardware (described in Section 3.1) that includes SMP Linux NFS clients connected to a prototype Network Appliance F85 filer via gigabit Ethernet.

2.1. Client performance issues

NFS is based on a “client makes right” design: the client is responsible for ordering bytes, managing network and server congestion, and otherwise handling the complex issues of implementing a distributed file system. This leaves the server simple and scalable [15]. In fact, NFS servers maintain very little state. Satyanarayanan, *et al.* [16] justify this architecture by pointing out that

¹ This document was written as part of the Linux Scalability Project at CITI, U-M. The work described here was supported via a grant from Network Appliance, Incorporated.

in typical client/server distributed systems, "workstations have cycles to burn." Consequently, an NFS client tends to be complex, which can interfere with efficiency and correct behavior.

Measuring NFS server performance is well understood. Computer science literature contains many examples of benchmarks meant to quantify NFS system performance or server performance [13, 17, 23]. SPEC SFS is a typical NFS server benchmark [19]. To remove client behavioral and performance variations from benchmark results, SPEC SFS uses its own user-space NFS client to access NFS servers under test.

Client performance measurement differs from server performance measurement. Generic file system benchmarks are biased towards exercising the weaknesses of disk storage, which is not terribly useful in divining the nature of a file system implementation that uses a network device as its back end. For example, *iozone*, a typical file system benchmark, tests both random and sequential read and write requests [13]. A client sends random write requests to a server as fast as it sends sequential ones. If performance differences exist between random and sequential NFS accesses, it is likely we are measuring server disk performance and not client behavior.

NFS client performance depends on the performance of networks and servers. It is problematic, however, to operate an NFS client without any server, thus it is difficult to isolate performance problems specific to a client. A slow server or network can cause application performance problems that are relatively easy to identify and fix; as we demonstrate, *faster* server performance can also degrade client performance.

A client's on-the-wire write request behavior sometimes affects server performance and scalability. Clients can modulate an application's unfortunate (random) access pattern to help servers scale better [7, 9]. The relationship between client and server must be carefully considered when dissecting client performance issues. In this paper, we focus only on a client's ability to get requests to the server. In later work, we may approach issues of server scalability that arise from client misbehavior.

One way to measure client performance is to eliminate performance bottlenecks from downstream components, using fast networking technologies and non-volatile RAM on the server, and to push the client as hard as possible to see what breaks. Just as SFS uses the same client to test different servers, a simple memory-based server could be developed to compare clients more fairly.

Another approach compares the performance and behavior of a single client under more typical workloads across a variety of networking conditions and server types. For both approaches, it is necessary to be wary of the bias of traditional file system benchmarking towards measuring disk behavior instead of other factors that are more important to client performance.

We borrow from both approaches in this study. Our hardware test bed consists of high-performance SMP Linux client hardware connected via a high-performance gigabit Ethernet switch to a prototype Network Appliance F85 filer. Also included in our test bed are a four CPU Linux server, and several single-CPU Solaris NFS clients (not used in this report). Comparing behavior and performance among these clients and servers exposes performance issues that might otherwise escape attention.

2.2. Related work

Little related work focuses specifically on NFS client performance. Improving NFS performance often amounts to helping the server use its disks more efficiently by improving client caching strategies, as in Dahlin, *et al.* [3]; or as in Juszczak, who adds write clustering to clients to help server scalability [7]; or by adding new features to the protocol, as in Macklem's Not Quite NFS [9].

Martin and Culler investigate NFS behavior on high performance networks, but do not address implementation specific issues in existing clients [10].

The closest previous work we found describes performance improvement (reduced CPU loading) through elimination of data copies in the 4.3BSD Reno NFS implementation [8].

2.3. Inter-run variance on Linux

Our experience with performance measurement on Linux has taught us to expect large variations in results between individual benchmark runs on the same O/S version and software and hardware configurations.

Other benchmarks performed by the authors in the past have revealed inexplicable variations in performance of several parts of the Linux kernel, including the virtual memory subsystem, the scheduler, and parts of the system whose correctness depend on the global kernel lock. There are often one or more outlying data points that skew average results, often masking relevant behavior. Such variations are not common in commercial operating systems such as Solaris. The best

results on Linux are excellent, but they are too often hampered by the outliers, leaving only moderate to good performance on average. Several measurements reported here illustrate this phenomenon.

To make forward progress we must often ignore these variations. Over time, our experience resolves many of these issues, but one could wish that untuned system behavior were more consistent.

To address this, we generally report single run results in this paper. The “shape” of the results is typically consistent from run to run, including any highly variable outlying results. We are most interested in trends rather than precise measurements, noting any anomalies.

2.4. Introducing our simple write benchmark

We started by measuring the Linux NFS client with Bonnie to understand several aspects of Linux client performance in combination, under a simple but typical load. We refined our benchmark to include only a small part of the suite of tests performed by Bonnie. In this section we discuss what was left out, and why.

Using a simple microbenchmark rather than a complex application simulation provides immediate and uncomplicated feedback without the additional effects of other application processing, improving the repeatability of results. It also offers a workload that drives specific components of a client with surgical precision. However, a microbenchmark does not offer a clear assessment of real world application performance impact.

We based our benchmark program on the block sequential write portion of the Bonnie file system benchmark. This test measures how quickly an application can write 8 KB chunks into a fresh file. Writing into a fresh file narrows our focus to write code pathways because the client does not read any preexisting file data from the server to complete write requests. Write throughput depends on the behavior of the kernel’s VM, networking, and RPC layers, and offers a generic picture of file system performance. In addition, raw write performance is important to many typical real world workloads.

Both read and write operations are network-intensive because data is transmitted along with these requests. However, client O/S caching moderates the performance of application read requests on the client; writes reflect network efficiencies and latencies more directly [14]. Using sequential writes we minimize disk latency (*i.e.*, seek time) on typical disk-based servers. As pointed out in Section 2.1, we gain little new information about a client by comparing random and sequential results. We considered testing against a memory-only

server, but we chose to start with a benchmark that does not require atypical server modifications. Thus we have a simple and typical application to run on a client that exercises many of the critical paths between client and server.

Bonnie includes the final `close()` call in elapsed time and throughput calculations to capture I/O that occurs after the last `write()`. However, for many local file systems, dirty data remains in the system’s data cache after the final `close()` operation. To make fair comparisons between NFS (which always flushes completely before last close due to close-to-open semantics) and local file systems (which may delay flushing to improve perceived performance), our benchmark reports three throughput results: one for all writes, one for the subsequent flush operation, and one for the final close operation. Each result is a throughput measurement reported in megabytes per second (MBps), and is calculated by dividing the total number of bytes written by the amount of time from the beginning of the benchmark until just after the respective operation (writes, flush, close).

Our benchmark also reports system call latency. One can calculate throughput by dividing average system call latency into the average byte size of each request. Reducing system call latency has immediate positive effects on throughput. However, to get to the heart of system call misbehavior, it is sometimes necessary to record *actual*, and not *average* latency. As we demonstrate, jitter (variation in latency from one call to the next) drastically degrades data throughput in our test, and is easily revealed when examining actual results rather than computed averages.

3. Write latencies in the Linux NFS client

Here we report the results of our benchmark when run on an SMP Linux client against files on a Linux NFS server and a Network Appliance filer. Our goal is to identify and correct write performance problems.

The first section describes our software and hardware configuration, and the following sections report our measurements and describe our fixes. We finish with a description of recent improvements to the Linux NFS client resulting from our work.

3.1. Systems under test

In this section, we describe the systems used during these tests.

Client system: Our client software runs on a dual processor Pentium III system based on the ServerWorks III LE chipset. The processors are 933 MHz FC-PGA packages with 256 KB of level 2 cache. The front-side bus and SDRAM speed is 133 MHz. There is 256 MB of PC133 registered SDRAM in each system. The client has one 30GB IBM Deskstar 70GXP EIDE UDMA100 drive. Because of limitations in the ServerWorks south bridge, the IDE controller runs in multiword DMA mode 2. The ServerWorks chipset supports two 64-bit/66 MHz PCI slots; there is a Netgear GA 620T gigabit Ethernet NIC in one of these that supports 1000base-T (copper). The Netgear card uses the Alteon Tigon II chipset. This system runs a Linux 2.4.4 kernel with the Red Hat 7.1 distribution.

NetApp filer: The Network Appliance filer is a prototype F85 with eighteen 36 GB Seagate 336704LC SCSI drives. The F85 has a single 833 MHz FC-PGA Pentium III with 256 KB of level 2 cache, 256 MB of RAM, and 64 MB of NVRAM on a PCI card. The system supports several 64-bit/66 MHz PCI slots that contain a Q-Logic ISP 1280 SCSI controller and a fiber optic gigabit Ethernet card based on Intel's GbE chipset. Data stored on this system is contained in RAID 4 volumes. This system runs a pre-release of Network Appliance's DATA ONTAP operating system². Special options enabled on the test volume include the `no_atime_update` option, which eliminates seek-intensive inode write activity during workloads that consist mostly of read requests. This option probably has no effect for our write-intensive workloads. The test volume contains eight disks in a single raid group. Snapshots are enabled during these tests.

Linux server: Our Linux NFS server is a four-way Intel system based on the i450NX mainboard. There are four 500 MHz Katmai Pentium III CPUs, each with 512 KB of level 2 cache. The front-side bus and SDRAM speeds are 133 MHz. The system contains 512 MB of SDRAM and six Seagate SCSI LVD drives of varying model, controlled by a Symbios 53c896 SCSI controller. The system is network-connected via a Netgear GA 620T 1000base-T Ethernet NIC installed in a 32-bit/33 MHz PCI slot. This system runs a Linux 2.4.4 kernel with the Red Hat 7.1 distribution. NFS files stored on this system reside on a single physical disk (no RAID). To maximize server write performance, we use the `async` export option; throughput results reported for the Linux server are therefore not comparable to a production server.

² Benchmark results produced on prototype hardware and software do not necessarily reflect the performance of any released product.

These systems are connected to a single Extreme Networks Summit7i Ethernet switch. The copper connections are made via CAT6 UTP cabling, and the fiber connection to the filer is standard multi-mode. Jumbo packets are not enabled on the switch or on any of the systems under test during these benchmarks. Unless otherwise mentioned, all network links are one Gbps, full duplex.

Both the Network Appliance filer and the Linux NFS server are mounted with typical mount options: NFS version 3 via UDP, `rsize=wsize=8192`. As of kernel 2.4.4, the Linux kernel NFS server does not support sizes larger than 8K. Using a smaller `wsize` causes the Linux client to use only synchronous network writes, resulting in a significant drop in write throughput. In addition, these sizes match the block size of our simple write benchmark.

The Network Lock Manager is disabled during our testing to reduce protocol overhead. Later we can test how much overhead is caused by Lock Manager interaction, after we quantify the baseline overhead for data transfer.

Using jumbo Ethernet frames is an easy optimization that can improve data throughput. However, jumbo frames work only for networks that allow large frame sizes from end to end, which makes them unsuitable in many situations. Because many realistic local and wide-area networks use smaller frames, it is useful to study the cost of packet fragmentation and reassembly. Thus, we chose to leave jumbo frames disabled during our tests.

3.2. Local versus network write performance

To begin, we compare the performance of sequential writes into a local file system (`ext2fs` [2] on the client) to the performance of sequential writes into a networked file system (NFS served from the filer and from the Linux NFS server). This compares the cache performance of `ext2` against the cache performance of NFS *without regard to back end performance*. `Ext2` cached write performance is a target for NFS client cached write performance.

This test calculates write throughput by dividing the total number of bytes written by the elapsed time required for all of the `write()` system calls to complete. Figure 1 shows throughput results that include only write calls, not including the final `flush()` and `close()` calls included. To allow a better comparison, the latter results are not included because `ext2fs` usually does not flush after `close()`.

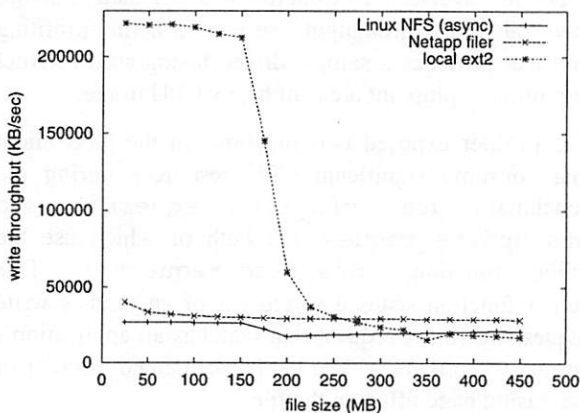


Figure 1. Local v. NFS cached write performance.

Write throughput is measured for test files between the sizes of 25 MB and 450 MB. Note that the large peak in memory write performance for local files does not appear for NFS files. NFS memory write throughput remains constrained to network/server throughput.

Writes to local files are very fast while there is still memory available to cache dirty data. As the test file size approaches the size of client memory, performance drops to raw disk speeds. In contrast, the NFS client constrains write throughput even though there is ample memory available to cache writes. During the test, the application can generate data only as fast as the NFS server can take it, no matter how small the file is; little or no write buffering appears to occur on the client. In the next subsection, we explore this limitation.

3.3. Periodic latency spikes

Early in our testing we discovered that `write()` system call latency varies wildly but periodically. To explore `write()` system call latency, we execute our benchmark against a single 40 MB file residing on the Network Appliance filer, and report latency for `write()` system calls during the test. A typical result is shown in Figure 2.

While most writes complete within 300 microseconds, there is a periodic jump in latency approximately every 85 system calls. The latency for these slow system calls is over 19 milliseconds. While there are relatively few of these slow calls (37 out of 2560 calls in this run, or about 1.5%), they inflate the mean latency for the run from 139.6 microseconds per call (excluding the 37 calls exceeding 1 millisecond) to 482.1 microseconds per call, a factor of almost 3.5.

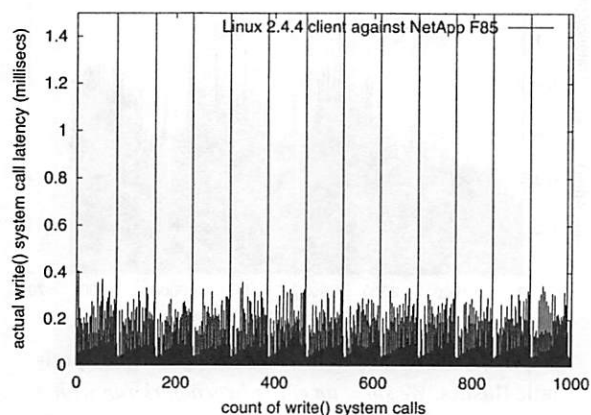


Figure 2. Write() system call latency. This figure shows the first 1000 write system calls during a 40 MB benchmark run. Periodically, write system calls take more than 19 milliseconds, increasing the mean latency, thus decreasing overall throughput.

We observe similar results with both the Network Appliance filer and the Linux NFS server. The latency spikes do not appear in write requests on the wire.

Eliminating spiky latency behavior seems likely to lower average write latency and improve write throughput. We instrumented the Linux NFS client's write code path to record the time required for each step of a `write()` system call. We use the Linux kernel's `do_gettimeofday()` kernel function to capture wall clock time on either side of a target section of code, then record the timings in the kernel log.

We discovered several places where the Linux NFS client delays writer threads to keep memory usage in check. It delays writers when the number of pending write requests for an inode or mounted file system exceeds fixed limits. When the per-inode request count grows larger than `MAX_REQUEST_SOFT` (whose value is 192 in the 2.4.4 kernel) the NFS client forces the writer thread to schedule all pending writes for that inode and wait for their completion before resuming the current request. When the per-mount request count grows larger than `MAX_REQUEST_HARD` (whose value is 256 in the 2.4.4 kernel) the NFS client suspends any thread writing to that file system until another thread signals there are fewer than `MAX_REQUEST_HARD` requests. Each internal write request is no larger than a page. This implementation does not employ hysteresis to smooth the request load.

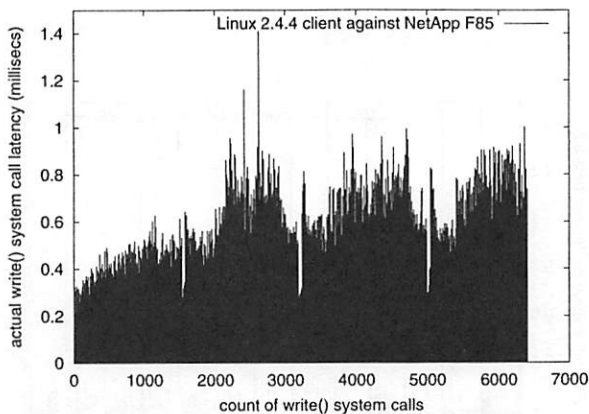


Figure 3. Write() system call latency without periodic flushes. We show an entire benchmark run with a 100 MB file. The latency axis is the same as Figure 2. The periodic spikes in write system call latency are gone, but average latency grows worse over time.

These limits prevent a large backlog of write requests. This is a classic time-space tradeoff. By limiting the amount of space available to buffer NFS writes the operating system avoids the expense of reclaiming pages at network and server latency speeds. It also avoids memory starvation if many write requests are pending when a server becomes unavailable.

Every system call in our test generates two write requests (8192 bytes is two pages, thus two requests). After the test application makes 90 write() calls, at least 180 internal requests are queued on the test file's inode. If the server is lagging, there may be requests from writes older than the past 90 system calls. Therefore, every 80 to 90 system calls, the client flushes the inode's write request queue. This produces the spiky latency seen in Figure 2.

In the Linux NFS client, a separate daemon, called `nfs_flushd`, flushes cached write requests behind a writing application. To minimize the cost of writes, the client should cache as many requests as it can in available memory [9]. The Solaris NFS client, for example, flush write requests only when the application requests it (via `fsync()` or `close()`), or unless the client cannot allocate more memory for new requests, in which case the VFS layer blocks the writer [4].

We see in Figure 3 that the periodic latency spikes are gone. However, mean latency does not improve: for the entire run (6400 writes in this case) the average latency is 484.7 microseconds. Furthermore, latency increases over time. We investigate this behavior in the next section.

3.4. List scans and sequential write performance

Scalability problems are often the result of lengthy data structure traversals. To establish whether data structure traversal limits throughput, we used a kernel-profiling tool that provides a sample-driven histogram of kernel execution to pinpoint areas of heavy CPU usage.

The profiler exposed two functions in the NFS client that consume significant CPU resources during the benchmark run: `nfs_find_request()` and `nfs_update_request()`, both of which use the inline function `_nfs_find_request()`. This helper function scans a sorted list of an inode's write requests to find a request that matches an application's current write request. The list is maintained in order of increasing page offset in the file.

Eliminating periodic write request flushing makes this per-inode list much longer. The sequential benchmark causes the client to traverse the list completely during each write system call, only to find no matching request, whereupon the client adds the new request to the end of the list.

To improve scalability, we implemented a hash table, similar to other hash tables in the Linux kernel, to manage the client's outstanding write requests. This hash table *supplements* the per-inode write request list. Finding a pending write request is now much faster, at a memory cost of eight bytes per request and eight bytes per inode, plus the size of the hash table itself.

The Linux VFS layer passes write requests no larger than a page to file systems, one at a time. Before the NFS client builds an RPC request, it maintains these page write requests on a per-inode list, ordered by page offset. Our modification inserts requests into a hash table based on the requesting inode and the page offset of the request.

All requests to the same page in the same inode are kept in the same hash bucket, so any overlapping requests are detected by searching all the requests in a single bucket. The client usually caches only a single write request per page to maintain write ordering, so this is normally not an issue. Write requests are coalesced into `wsize` chunks just before the client generates write RPCs.

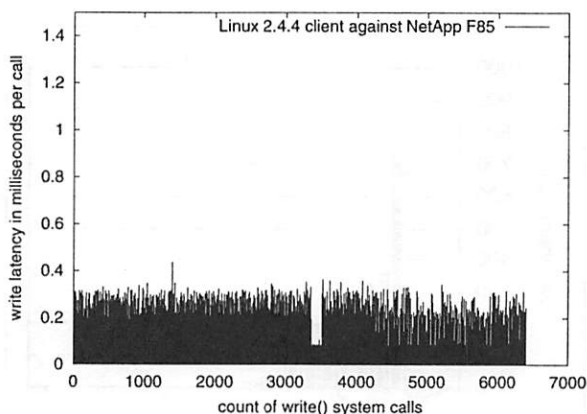


Figure 4. Write() system call latency with scalable data structures. Write latency remains low even as the number of outstanding requests increases for the entirety of this benchmark run against a 100 MB file. For comparison, the latency axis is the same as in Figures 2 and 3.

We see the improvement of using a hash table to track write requests in Figure 4. Write system call latency during this run averages 136.9 microseconds per call, about the same as the mean for the original 2.4.4 client when latency spikes are excluded (see Figure 2). The sustained memory throughput of our sequential write benchmark is now almost 115 MBps, compared to 28 MBps in Figure 1 for a 100 MB file.

We also notice a gap of greatly reduced jitter for a few hundred calls in the middle of Figure 4. This gap appears in several runs against the filer. We investigate this further in the next section.

3.5. Global kernel lock on SMP hardware

Having eliminated the extra flush in the write path, and implemented a scalable hash table to track write requests, we now compare write throughput performance of our client against a Network Appliance filer and against a four-way Linux NFS server.

During a typical run of our write benchmark with a 5 MB file, the filer sustains about 38 MBps of network throughput. Our benchmark reports it can generate about 115 MBps of writes. On the other hand, the Linux server can sustain only 26 MBps of network throughput, less than 70% of the filer's network throughput, yet our benchmark writes at a rate greater than 138 MBps 20% faster than the filer run.

To explore this unexpected behavior, we again examine write latency. Figure 5 shows a histogram of write() system call latencies. While some of these calls take less than 100 microseconds, many take longer. The distribution shows there are more slow calls when the file resides on the faster of the two servers.

Surprisingly, the client requires less overhead to buffer writes when it is sending data to a slow server. We verified this result with a server on 100 Mbps Ethernet. The benchmark writes to memory even faster with this server, which sustains less than 10 MBps per second of network throughput.

Kernel execution profiling shows that, during benchmark runs, the global kernel lock taken in `nfs_commit_write()` is under contention on SMP hardware. The lock section is the fourth largest CPU consumer in the kernel, exercised more than twice as often as the fifth largest consumer. A profile analysis of this section shows that the lock taken in `nfs_commit_write()` is the only contributor to CPU time sampled in the lock section.

On SMP hardware, even a single writer thread uses more than one CPU, because data that is not flushed during a write() system call is flushed later by the NFS client's write-behind daemon, `nfs_flushd`. Kernel lock contention results when both the single writer thread and the flush daemon generate network write requests. `nfs_flushd` holds the global kernel lock whenever it is awake and flushing requests. We suspected the flush daemon was causing contention, but after removing the global kernel lock from the daemon, we found little improvement.

Next we instrumented the write path to find out where the most time is spent, and found that the kernel spends 50 microseconds per write request in the network layer (`sock_sendmsg()` is called from the RPC layer for each RPC request). This accounts for almost 90% of the time per request spent waiting in the NFS client's write path to acquire the kernel lock.

During the development of the Linux 2.3 kernel, the global kernel lock was removed from Linux's network implementation. Because it is now no longer necessary to hold the kernel lock while calling the network layer, we release the lock then reacquire it when `sock_sendmsg()` returns. This permits other writing processes to make progress while the network layer sends the current request.

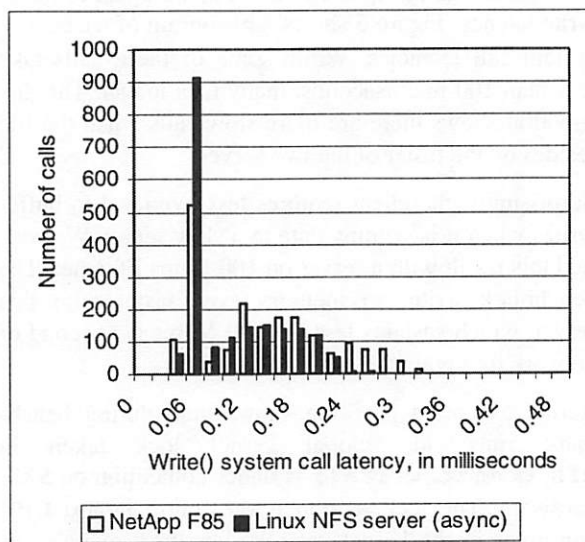


Figure 5. Write() system call latency against different servers. This figure shows the latency of write calls during a benchmark run against a 30 MB file. Both runs have about the same minimum latency, but the filer run has a large number of lengthy calls. The average latency of client memory writes increases when a file is stored on a faster server.

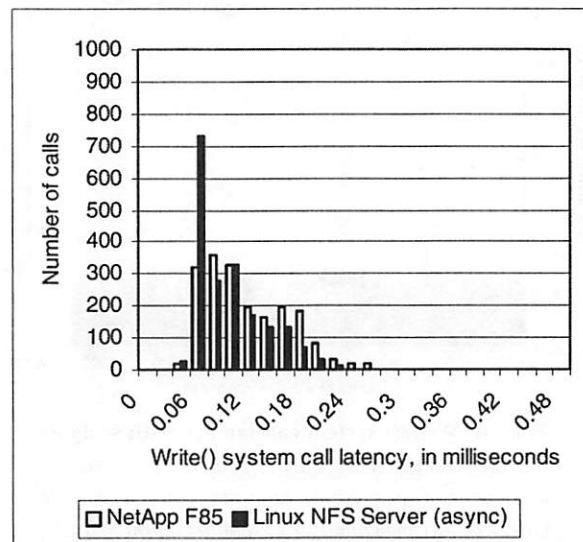


Figure 6. Write() system call latency with less lock contention. This figure shows that maximum latency and latency variation (jitter) are clearly reduced. On average, filer writes still take longer than writes to the Linux NFS server, but the difference is much smaller. Minimum latency remains roughly the same, suggesting that latency variation, in this case, is the result of lock contention.

Figure 6 shows the improvement in write() system call latency that occurs after removing the kernel lock around `sock_sendmsg()`. During this run, our calculated results also improve: the mean write() system call latency drops for both benchmark runs on the new client (127 versus 149 microseconds for the filer, 105 versus 113 microseconds for Linux), and the filer's maximum latency also drops, from 381 microseconds to 292 microseconds.

In calculating these averages, we excluded the first data point in all four runs. The latency for the first write() system call was almost a millisecond during two of the runs.

We note that the minimum latency hardly changes. This agrees with the idea that the latency variation is not a code path issue, but results from the writer waiting to acquire a resource, such as a lock.

We ran our 5 MB benchmark with the lock modification. Against the filer, the benchmark runs at almost 140 MB per second, better than a 20% increase over the earlier 115 MB per second. The benchmark runs at 147 MB per second against the Linux server, a 6.5% improvement. Lock contention measured by the profiler is

almost entirely gone. These results are summarized in Table 1 (overleaf).

Although most of the unexpected inversion of performance is gone, the client still runs 5% faster against a server that is 40% slower. We discuss this further in the next section.

3.6. The cost of responding to server replies

Even though the Network Appliance filer provides better network throughput than the Linux NFS server, applications writing to the filer run slower. Despite the fact that less client processing is required for filer writes, which don't require an additional COMMIT RPC, client throughput to a fast server is hampered by lock contention and the cost of handling server replies at a higher rate. A faster server forces the client to do the same amount of work in less time. This explains the unexpected inversion of client performance.

	<i>Network</i>	<i>Lock</i>	<i>No lock</i>
NetApp F85	38 MBps	115 MBps	140 MBps
Linux (async)	26 MBps	138 MBps	147 MBps

Table 1. Application write throughput, before and after lock modification. *Network write throughput is compared to application write throughput when writing a 5MB file. Removing the global kernel lock from the RPC layer improves cached write throughput for files residing on both the Network Appliance filer and the Linux NFS server. Section 3.6 explains why applications can write faster to a slower server.*

Tests with a single application writer thread contending with a single flusher thread show less than ideal scaling. On a client with a single CPU, we expect to find the flusher thread taking some CPU time away from a user-level writer thread, increasing as server throughput increases. On a client with more than one CPU, however, the writer thread and the flusher thread should not interfere. We suspect that faster servers will exacerbate on SMP Linux clients until this issue is addressed.

Recall the short period in Figure 4 during which `write()` system call latency is much lower on average. This can now be explained by reduced SMP lock contention and interrupt load when the filer briefly stops responding to network write requests during a file system checkpoint [5]. In effect, the filer behaves like an infinitely slow server during this period, momentarily eliminating SMP lock contention on the client. While the flusher thread is blocked, only the application writer thread is active. Other threads do not compete with the writer, allowing the client to return control quickly to the application. In other words, the difference between the slowest and fastest writes in Figure 4 is due to the client's cost of responding to server replies.

Moreover, fast networking introduces significantly increased interrupt loads. The new network device driver API ("NAPI") in Linux 2.5 may help here, especially on single processor systems, by improving system behavior during intense interrupt loads that can result when a client is communicating with a high-performance server over a low latency network. When the system recognizes that a device is producing interrupts at a high rate, it masks the device interrupt and polls instead. As the workload decreases, the interrupt is re-enabled to keep I/O latency reasonable. This tech-

nique is further expanded by Mogul and Ramakrishnan [12].

In future work, we hope to explain why the network layer takes more than 50 microseconds per RPC request on a 933 MHz processor. We suspect IP fragmentation is a major expense. Jumbo frames, a feature of gigabit Ethernet, may help by reducing the need for fragmenting and reassembling large RPC requests in the IP layer, although this does not extend to WANs, in general.

Removing the global kernel lock from the write path yields considerable improvements in throughput and application concurrency. As it happens, the RPC layer also acquires the global kernel lock to ensure the integrity of its internal data structures. Removing the global kernel lock from the RPC layer will allow an SMP system with multiple network interfaces to process more than one RPC request at a time, allowing concurrent writes to separate files and to separate servers from separate client CPUs.

3.7. Final measurement

Figure 7 illustrates how our modifications have improved client write performance. With our modifications, NFS write performance is very good while memory is available to buffer write requests, but drops to the server's throughput rate when the client exhausts memory.

The left side of Figure 7 shows that memory write performance to NFS files is considerably improved. Write performance is no longer limited to network and server speeds. Client scalability defects continue to cause memory writes to files on the Network Appliance filer to be 7 MBps slower than to files on the Linux NFS server. The right side of Figure 7 shows that as client memory is exhausted, the filer sustains greater network write throughput than the Linux NFS server can.

NFS write performance is still not as good as writes to local files, however. We believe this is due to the costs on the client of responding to the server's replies. These costs, which include interrupt handling and network processing, are clearly greater than simply managing disk I/O.

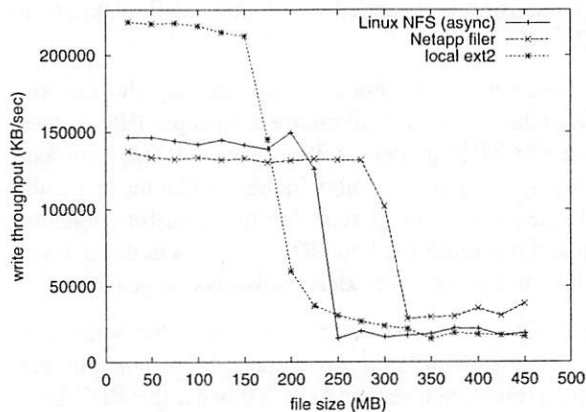


Figure 7. Local v. NFS cached write performance, revisited. This figure shows write throughput for test files between the sizes of 25 MB and 450 MB. NFS write throughput is considerably improved compared to Figure 1. Application write throughput no longer tracks network write throughput for small NFS files. Maximum cached write throughput is nearly the same against both servers.

Throughput for the local test and the test against the Linux NFS server immediately trail off for file sizes that exceed the physical memory size of the client, but the benchmark is able to sustain high data throughput longer when the test file resides on the Network Appliance filer. We conjecture that the filer's NVRAM acts as an extension of the client's page cache, allowing writes to the server to proceed at near local memory speed until the server's NVRAM is full. The fact that the filer is able to process requests faster also makes more client buffers available for a little while.

With workloads that hold a file open for a long time and write asynchronously (that is, without the requirement that data be made permanent before the `write()` system call is complete), the slower Linux NFS server has a slight advantage. Where applications write then immediately flush or close, or where applications require data permanence before a `write()` system call returns (e.g. databases), the Network Appliance filer, with its greater network and disk throughput, performs better. Though cached writes are slightly slower on the client, applications regain control sooner after they flush or close a file when writing to a faster server. As client scalability improves, applications can take advantage of improved memory write throughput and better network throughput.

3.8. Recent releases of the Linux NFS client

After writing the initial drafts of this paper, we shared our work with Trond Myklebust, the maintainer of the Linux NFS client. Trond built on our ideas, creating a safe version of our patch to remove the global kernel lock from the RPC layer and the NFS client's write path. This patch is available on his web site in the experimental patches section [22].

Trond also made a simple change to the write request queuing logic to reverse the order of the list, based on the results of our hash table experiment, to allow sequential writes to insert new requests into the request list in constant time, rather than walking the entire list. Finally, based on this paper and other recent work at Network Appliance, he replaced the flushing logic described in Section 3 with an entirely new system. This work now appears in Linux kernel releases following 2.4.15. Because so many other changes have occurred since the 2.4.4 kernel, a direct comparison is not meaningful. However, we hope to analyze some of these improvements in future work.

4. Discussion and future work

In this paper, we describe a simple sequential write benchmark to measure file system write latency and throughput. We show how this benchmark reveals performance and scalability problems in the Linux NFS client, and we describe several modifications to the Linux NFS client that improve application write latency and throughput.

4.1. Observations on client benchmarking

An NFS server's job is to store data and metadata in an organized way, and to move data between network cards and disks as efficiently as possible. Measuring these behaviors is well understood. On the other hand, a client's role is to translate and adapt the NFS protocol to its local environment efficiently. This is a much more subtle task.

Because a client is complex and completely dependent on the performance of servers and their disks, we use a microbenchmark, rather than a large suite of tests, to focus analysis on small parts of client behavior. As a result of our studies, we have identified several areas where client implementation directly affects application throughput. Some of these areas are already documented by previous work.

Networking efficiency

Packet fragmentation and reassembly, handling packet loss, eliminating data copies, handling heavy interrupt loads, and optimizing the number of network requests all contribute to the cost of handling server replies.

Caching efficiency

Effective caching makes NFS clients perform almost as well as local file systems. This means making the best use of available memory, as well as properly implementing cache coherency.

I/O scheduling

Unfortunate write scheduling can decimate application performance and server scalability.

Lock contention

With any number of CPUs, avoiding lock contention is critical. This has direct bearing on how well client performance scales when adding more CPUs and network interfaces.

Data structure efficiency

As the power of clients and the amount of cached data grows, it is vital to manage both efficiently.

Our current efforts focus on developing a suite of microbenchmarks of these aspects, in the style of McVoy's *lmbench* [11].

4.2. Future work

In this paper, we identified several specific issues with the Linux client that deserve further investigation. As our work continues, we hope to evolve benchmarks that measure each of these areas.

We want to assess further the impact of the global kernel lock on the scalability of the Linux NFS client. We also want to continue investigating why slower servers allow faster memory write throughput on Linux NFS clients, and why, in general, there continues to be so much variance between benchmark runs on Linux.

We especially want to prove our comparative methodology within real application domains. To keep our study on point we have focused mainly on our microbenchmark; future work will determine the real world impact of these changes. These techniques are also valuable for surveying NFSv4 client implementations [18]. Finally, we hope to explore improvements to the Linux NFS client that affect its behavior in corner cases that face advanced deployments outside the research

lab, such as its file locking and specialized caching behavior, and its performance with databases and massively parallel applications combined with network-attached storage.

Acknowledgements

The authors gratefully acknowledge the assistance of our colleagues Andy Adamson, Kendrick Smith, James Newsome, and Steve Molloy at the University of Michigan; Brian Pawlowski and Sudheer Miryala at Network Appliance; Spencer Shepler and Sun Microsystems; and especially Trond Myklebust for his helpfulness and thorough work on the Linux NFS client. Special thanks also to FreeNIX reviewers and to our shepherd, Jim McGinness. The Intel Corporation loaned CITI hardware used in this study.

The source for our simple write benchmark and a patch against Linux kernel 2.4.4 that includes the modifications discussed in this paper are available at the CITI web site:

<http://www.citi.umich.edu/projects/nfs-perf/patches/>

References

1. Bray, T. Bonnie Source Code. Netnews Posting, 1990.
2. Card R., Ts'o, T., Tweedie, S. "Design and Implementation of the Second Extended Filesystem." *Proceedings of the First Dutch International Symposium on Linux*, December 1994.
3. Dahlin, M., Mather, C., Want, R., Anderson, T., Patterson, D. "A quantitative analysis of cache policies for scalable network file systems." *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.
4. Eisler, Michael. Personal communication, September 2001.
5. Hitz, D., Lau, J., and Malcolm, M. "File System Design for an NFS File Server Appliance." *USENIX Technical Conference Proceedings*, January 1994.
6. "HTTP: A protocol for networked information." W3C, 1992. www.w3.org/Protocols/HTTP/HTTP2.html
7. Juszczak, C. "Improving the Write Performance of an NFS Server." *USENIX Technical Conference Proceedings*, January 1994.
8. Macklem, R. "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol." *USENIX Technical Conference Proceedings*, January 1991.

9. Macklem, R. "Not Quite NFS, Soft Cache Consistency for NFS." *USENIX Technical Conference Proceedings*, January 1994.
10. Martin, R., Culler, D. "NFS Sensitivity to High Performance Networks." *SIGMETRICS '99/PERFORMANCE '99 Joint International Conference on Measurement and Modeling of Computer Systems*, May 1999.
11. McVoy, L., Staelin, C. "Imbench: Portable Tools for Performance Analysis." *USENIX Technical Conference Proceedings*, June 1996.
12. Mogul, J., Ramakrishnan, K. "Eliminating Receive Live-lock in an Interrupt-driven Kernel." *USENIX Technical Conference Proceedings*, January 1996.
13. Norcott, W., *et al.* IOzone benchmark. See www.iozone.org.
14. Ousterhout, J. and Douglass, F. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems." *Proceedings of the ACM Symposium on Operating System Principles*, 23, January 1989.
15. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D. "NFS Version 3 - Design and Implementation." *USENIX Technical Conference Proceedings*, June 1994.
16. Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and West, M. "The ITC Distributed File System: Principles and Design." *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.
17. Shein, B., Callahan, M., Woodbury, P. "NFSSStone A network file server performance benchmark." *USENIX Technical Conference Proceedings*, June 1989.
18. Shepler, Spencer, *et al.* "RFC 3010 - NFS Version 4 Protocol specification." IETF draft standard, December 2000.
19. Standard Performance Evaluation Corporation. SPEC SFS97. www.spec.org/osg/sfs97/.
20. Sun Microsystems, Inc. "RFC 1094 - NFS: Network File System Protocol specification." IETF Network Working Group. March 1989.
21. Sun Microsystems, Inc. "RFC 1813 - NFS: Network File System Version 3 Protocol Specification." IETF Network Working Group. June 1995.
22. Myklebust, Trond. Linux NFS client pages, 2002. See www.fys.uio.no/~trondmy/src/.
23. Wittle, M., Bruce, K. "LADDIS: The Next Generation in NFS File Server Benchmarking." *USENIX Technical Conference Proceedings*, June 1993.

A Study of the Relative Costs of Network Security Protocols*

Stefan Miltchev

miltchev@dsl.cis.upenn.edu
University of Pennsylvania

Sotiris Ioannidis

sotiris@dsl.cis.upenn.edu
University of Pennsylvania

Angelos D. Keromytis

angelos@cs.columbia.edu
Columbia University

Abstract

While the benefits of using IPsec to solve a significant number of network security problems are well known and its adoption is gaining ground, very little is known about the communication overhead that it introduces. Quantifying this overhead will make users aware of the *price* of the added security, and will assist them in making well-informed IPsec deployment decisions.

In this paper, we investigate the performance of IPsec using micro- and macro-benchmarks. Our tests explore how the various modes of operation and encryption algorithms affect its performance and the benefits of using cryptographic hardware to accelerate IPsec processing. Finally, we compare against other secure data transfer mechanisms, such as SSL, `scp(1)`, and `sftp(1)`.

1 Introduction

The increasing need for protecting data communications has led to the development of several protocols that provide very similar services, most notably data secrecy/integrity and origin authentication. Examples of such protocols include IPsec, SSL/TLS, and SSH[8, 2, 11]. While each of the protocols is based on a different set of assumptions with respect to its model of use, implementation characteristics, and supporting applications, they all fundamentally address the same problem, namely to protect the secrecy and integrity of data transferred over an untrustworthy network such as the Internet.

Securing the data while in transit is not sufficient by itself in building a secure network: data storage, key management, user interface, and backup security must also be addressed to provide a comprehensive security posture. These are often overlooked, yet are an essential

part of a secure system. In this paper, we aim to quantify the costs of specific mechanisms and clarify the options available to system and network architects. In particular, we wish to quantify the performance implications of using various security protocols that are either widely used (e.g., SSL and SSH) or are expected to be in wide use (e.g., IPsec).

Compared to other network security mechanisms, IPsec offers many architectural advantages. Firstly, the details of network security are usually hidden from applications, which therefore automatically and transparently take advantage of whatever network-layer security services their environment provides. More importantly, IPsec offers a remarkable flexibility not possible at higher or lower network layers: security can be configured end-to-end (protecting traffic between two hosts), route-to-route (protecting traffic passing over a particular set of links), edge-to-edge (protecting traffic as it passes between “trusted” networks via an “untrusted” one, subsuming many of the current functions performed by network firewalls), or in any other configuration in which network nodes can be identified as appropriate security endpoints. However, a perception of complexity¹ and reduced performance have acted as deterring factors in its deployment and use. The former point is being addressed by new APIs and refinement of administrative interfaces that make configuration and use of IPsec easier. However, the performance issue has not received adequate examination.

In this paper, we investigate the performance of IPsec using micro- and macro-benchmarks. Our tests are designed to explore how the various modes and encryption algorithms affect its performance, the benefits of using hardware accelerators to assist the IPsec cryptographic framework, and finally compare against other secure transfer mechanisms, such as SSL, `scp(1)`, and `sftp(1)`. We use the OpenBSD operating system as our experimental platform, because of its support for

*This work was supported by DARPA under Contract F39502-99-1-0512-MOD P0001.

¹In particular with respect to configuration tools, and PKI support.

cryptographic hardware accelerators and its native IPsec implementation[9].

2 System Architecture

In this section we briefly describe the OpenBSD IPsec and Kernel Cryptographic Framework architecture. Since the goal of this paper is not to discuss the implementation details, we refrain from going into too much depth.

2.1 IPsec

The IP Security architecture [8], as specified by the Internet Engineering Task Force (IETF), is comprised of a set of protocols that provide data integrity, confidentiality, replay protection, and authentication at the network layer. This positioning in the network stack offers considerable flexibility in transparently employing IPsec for different roles (*e.g.*, building Virtual Private Networks, end-to-end security, remote access, *etc.*). Such flexibility is not possible at higher or lower levels of the network stack.

The overall IPsec architecture is very similar to previous work [5] and is composed of three modules:

- The data encryption/authentication protocols [6, 7]. These are the “wire protocols,” used for encapsulating IP packets to be protected. They simply provide a format for the encapsulation; the details of the bit layout are not particularly important for the purposes of this paper.

Outgoing packets are authenticated, encrypted, and encapsulated just before being sent to the network, and incoming packets are decapsulated, verified, and decrypted immediately upon receipt. These protocols are typically implemented inside the kernel, for performance and security reasons. A brief overview of the OpenBSD kernel IPsec architecture is given in Section 2.2.

- A key exchange protocol (*e.g.*, IKE[4]) is used to dynamically establish and maintain Security Associations (SAs). An SA is the set of parameters necessary for one-way secure communication between two hosts (*e.g.*, cryptographic keys, algorithm choice, ordering of transforms, *etc.*). Although the wire protocols can be used on their own using manual key management, wide deployment and use of IPsec in the Internet requires automated, on-demand SA establishment. Due to its complexity, the key management protocol is typically implemented as a user-level process.

- The policy module governs the handling of packets on their way into or out of an IPsec-compliant host. Even though the security protocols protect the data from tampering, they do not address the issue of which host is allowed to exchange what kind of traffic with what other host. This module is in fact split between the kernel (which decides what level of security incoming or outgoing packets should have) and user space (making higher-level decisions, *e.g.*, which user is allowed to establish SAs and with what parameters).

For more details on their implementation in OpenBSD, see [3].

2.2 OpenBSD IPsec Implementation

In the OpenBSD kernel, IPsec is implemented as just another pair of protocols (AH and ESP) sitting on top of IP. Thus, incoming IPsec packets destined to the local host are processed by the appropriate IPsec protocol through the protocol switch structure used for all protocols (*e.g.*, TCP and UDP). The selection of the appropriate protocol is based on the protocol number in the IP header. The SA needed to process the packet is found in an in-kernel database using information retrieved from the packet itself². Once the packet has been correctly processed (decrypted, authenticity verified, *etc.*), it is re-queued for further processing by the IP module, accompanied by additional information (such as the fact that it was received under a specific SA) for use by higher protocols and the socket layer.

Outgoing packets require somewhat different processing. When a packet is handed to the IP module for transmission (in `ip_output`), a lookup is made in the Security Policy Database (SPD) to determine whether that packet needs to be processed by IPsec. The SPD in OpenBSD is implemented as an extension to the standard BSD routing table. The decision is made based on the source/destination addresses, transport protocol, and port numbers. If IPsec processing is needed, the lookup will also specify what SA(s) to use for IPsec processing of the packet (even to the extent of specifying encryption/authentication algorithms to use). If no suitable SA is currently established with the destination host, the packet is dropped and a message is sent to the key management daemon through the `PF_KEY` interface [10]. It is then the key management’s task to negotiate the necessary SAs. Otherwise, the packet is processed by IPsec and passed to `ip_output` again for transmission. The packet also carries an indication as to what IPsec processing has already occurred to it, to avoid infinite processing

²Specifically, the destination IP address, the 32-bit SPI field from the IPsec header, and the IPsec protocol (ESP or AH) number.

loops.

2.3 OpenBSD Cryptographic Framework

To improve the performance of the cryptographic operations in IPsec, we developed a framework for cryptographic services in OpenBSD that abstracts the details of specific cryptographic hardware accelerator cards behind a kernel-internal API. The details of the framework are beyond the scope of this paper. However, we give a brief description here so the reader has the proper context within which to consider our measurements.

Besides abstracting the API for accessing these cards, the framework was designed with these goals in mind:

- **Asynchronous operation:** The kernel should not have to wait until the hardware finished the requested operation.
- **Load balancing:** If multiple cryptographic accelerators are present, they should be utilized such that throughput is maximized.
- **No dependence on hardware:** If no hardware accelerators are present, the system should offer the same services (albeit at lower performance).
- **Application independence:** Although the framework was initially developed for use with IPsec, it should be possible to use it to accelerate other kernel operations (*e.g.*, filesystem or swap encryption) and user-level applications (*e.g.*, the OpenSSL library).
- **Support for public key operations.** This is currently work in progress.

Work on the framework is still in progress, but the main skeleton is present and has been in use with IPsec since OpenBSD 2.8.

The framework presents two interfaces: one to device drivers, which register with the framework and specify what algorithms and modes of operations they support; and one to applications (*e.g.*, IPsec), which create “sessions” and then queue requests for these.

Sessions are used to create context in specific drivers (selected by the framework based on a best-match basis, with respect to the algorithms used) and can migrate between different cryptographic accelerators (*e.g.*, if a card fails or is plugged out of the system, as may be the case with PCMCIA adaptors, or if a higher-priority session arrives). This is achieved by requiring that all necessary context is provided with every request, regardless of the fact that a session has been created (the context is kept at the application and inside the accelerator cards and is not cached by the framework itself).

Applications queue requests on sessions, and the cryptographic framework, running as a kernel thread and periodically processing all requests, routes them to the appropriate driver. Once the request has been processed, a callback function provided by the application is invoked, which continues processing (in the IPsec case, passes the packet to `ip_output()` for transmission). A software pseudo-driver registers with the framework as a driver of last resort (if any other driver can process the session, it will be preferred).

User-level applications (*e.g.*, the OpenSSL library or the SSH daemon) can access the hardware through the `/dev/crypto` device, which acts as another kernel application to the framework, using the same API. Public key operations are modeled in the same way.

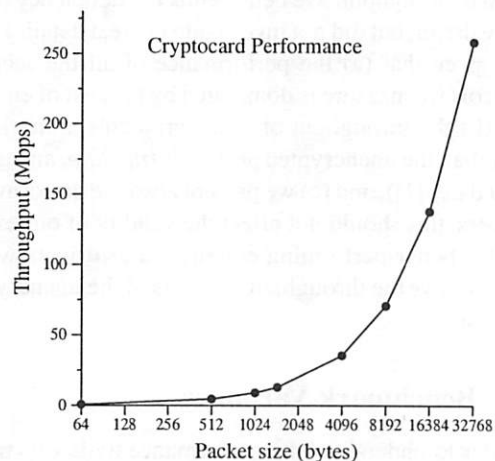


Figure 1: Cryptographic card performance.

Smart ethernet cards Although the cryptographic framework does not directly take advantage of ethernet cards that support IPsec processing offloading (since they are not general-purpose cryptographic accelerators), we extended the IPsec stack to use them. Unfortunately, at the time of writing this paper, driver support for these cards was not completed and thus we could not measure their performance. The cards of this type we are familiar with are 100Mbps full-duplex, and it seems reasonable (given our results with dedicated cryptographic processors) to assume that they can achieve that performance. Unfortunately, at the time this paper was written, we did not have enough information to write a device driver that could take advantage of such features.

3 Evaluation

Our test machines are x86 architecture machines running OpenBSD 3.0. More specifically, they are 1 GHz

Intel PIII machines with 256 MB of registered PC133 SDRAM, 10 GB Western Digital Protege IDE hard drives, Intel PRO/1000 F network adapters and some 3Com 3c905B 100Mbps network adapters. We chose Supermicro 370DE6 motherboards based on the ServerWorks Serverset III HE-SL chipset with dual PCI buses. Thus we were able to place our gigabit cards and crypto-cards on separate PCI buses. For some of our experiments we used the Broadcom 5820 crypto-cards. The manufacturer of these cards advertises 300Mbps 3DES; our own evaluation showed a peak measured performance of around 260Mbps, probably due to operating system overhead. We summarize our results in Figure 1. Notice that even in the best case (host-to-host, large socket buffers), we only get slightly over half the nominal throughput. We believe this is a deficiency in the device driver, but did not investigate in great detail. However, given that (a) the performance of all the security protocols we measure is dominated by the cost of encryption, (b) the throughput of those protocols is markedly lower than the unencrypted protocols (*ftp*, *http*, and unencrypted *ttcp*[1]), and (c) we present absolute performance numbers, this should not affect the validity of our experiments: better-performing ethernet cards/drivers would only improve the throughput numbers of the unencrypted protocols.

3.1 Benchmark Variables

In order to understand the performance trade-offs of using IPsec as well as how it compares to other approaches we designed a set of performance benchmarks. Our experiments were designed in such a way as to explore a multitude of possible setups.

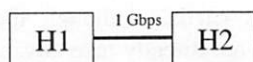


Figure 2: Host-to-Host topology.

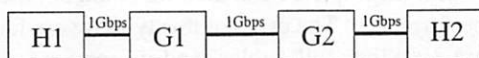


Figure 3: Host-to-Gateway-to-Gateway-to-Host topology. In this case experiments that use IPsec form a tunnel between gateways.

Our experiments take into consideration five variables: the type of utility used to measure performance, the type of encryption/authentication algorithm used by IPsec (or other applications), the network topology, use of cryptographic hardware accelerators, and the effects that the

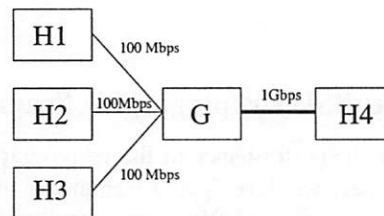


Figure 4: 3 Hosts-to-Gateway-to-Host topology. We use two IPsec tunnel configurations, end-to-end (where the 3 hosts form tunnels to the end host) and gateway-to-host (H4).

added security has on the performance of the system. For the IPsec experiments, we use manually configured SAs; thus, the performance numbers do not include dynamic SA setup times. For SSL, scp, and sftp, bulk data transfers include the overhead of session setup; however, that overhead is negligible compared to the cost of the actual data transfer.

Large filetransfer experiments were repeated 5 times, all other experiments were repeated 10 times and the mean was taken. Error bars in our graphs represent one standard deviation above and below the mean. Graphs presenting *ttcp* measurements do not show error bars to avoid clutter, however the standard deviation is small in all cases.

We will go into more detail about each experiment in the following section.

3.2 Micro-benchmark Results

In Figures 5, 6, 7 and 8, we explore different network configurations using the *ttcp* benchmarking tool. We explore how the various encryption algorithms affect performance and how much benefit we get out of hardware cryptographic support. The host-to-host topology is used as the base case, and should give us the optimal performance of any data transfer mechanism in all scenarios. The other two topologies map typical VPN and “road warrior” access scenarios.

The key insight from our experiments is that even though the introduction of IPsec seriously worsens performance, our crypto hardware improves its performance (relative to pure-software IPsec) by more than 100%, especially in the case of large packets. For the host-to-host experiment, we see that throughput over IPsec varies from 40% of the unencrypted transfer (for small packet sizes) to 30% (for 8KB packets³). We notice a similar situation in the VPN configuration (host-gateway-gateway-host). In the last two scenarios, the difference

³This is the size of the buffer that the *ttcp* benchmark is using for reading and writing to the network.

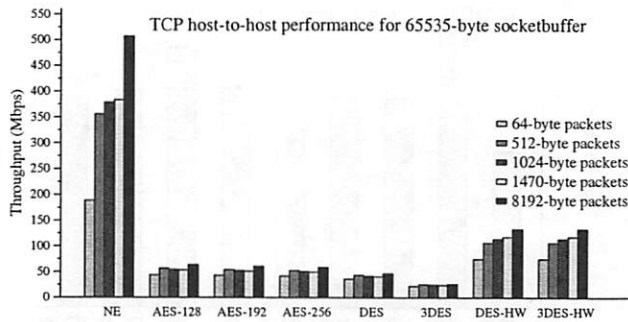


Figure 5: The `ttcp` utility over TCP, for the host-to-host network configuration with 65535 bytes of socket buffer. *NE* means No Encryption. We measure the AES algorithm with three different key sizes (128, 192, and 256 bits), as well as DES (56 bits) and 3DES (168 bits). The suffix “-HW” indicated use of a hardware accelerator for that cryptographic algorithm. In all cases where IPsec is used, we use HMAC-SHA1 as the data integrity/authentication algorithm; when hardware acceleration is used, HMAC-SHA1 is also accelerated.

in performance is less marked between the unencrypted and the hardware-accelerated cases, since the aggregate throughput of the three hosts on the left is limited to at most 300 Mbps (due to the topology).

In our experiments, we also noticed some anomalous behavior with 512 byte packet sizes, we believe that this has to do with buffer mis-alignments in the kernel and will investigate further in the future using profiling.

In our previous experiments we stress-tested IPsec by maximizing network traffic using `ttcp`. In our next set of experiments, we investigate how IPsec behaves under “normal” network load and how it compares with other secure network transfer mechanisms like `scp` (1) and `sftp` (1). Our tests measure elapsed time for a large file transfer in two different network configurations, host-to-

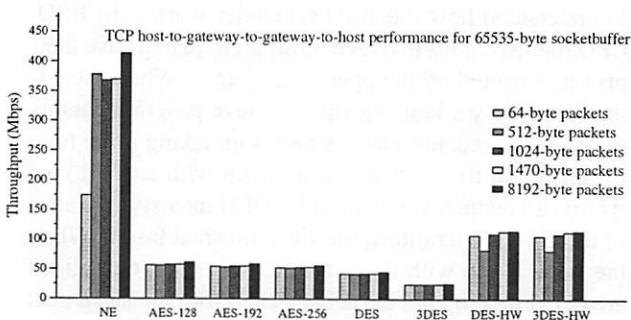


Figure 6: The `ttcp` utility over TCP, for the host-to-gateway-to-gateway-to-host network configuration with 65535 bytes of socket buffer. IPsec is used between the two gateways.

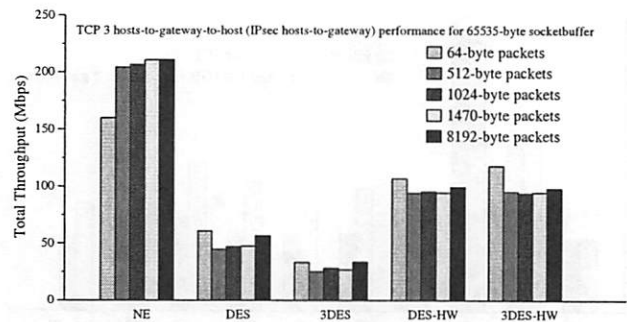


Figure 7: The `ttcp` utility over TCP, for the 3 hosts-to-gateway-to-host network configuration with 65535 bytes of socket buffer. In this case we create an IPsec tunnel between hosts H1, H2, H3 and the gateway.

host and host-to-gateway-to-gateway-to-host. In the first case, IPsec is used in an end-to-end configuration; in the second case, IPsec is done between two gateways.

Figures 9 and 10 present our results. Since we are doing large file transfers, we easily amortize the initialization cost of each protocol. Comparing the two figures, we notice that most of the time is actually spent by the file system operations, even after we normalize the file sizes. Another interesting point is that when we use IPsec the file transfer is quicker in the gateway network topology compared to the direct link. At first this might seem counter-intuitive, however it is easily explained: in the gateway case, the IPsec tunnel is located between the gateways, therefore relieving some processing burden from the end hosts that are already running the ftp program. This leads to parallel processing of CPU and I/O operations, and consequently better performance, since the gateway machines offload the crypto operations from the end hosts. Note that IPsec is not used for the plaintext ftp, scp, and sftp measurements.

Figures 11 and 12, compare IPsec with `ssl` (3) as

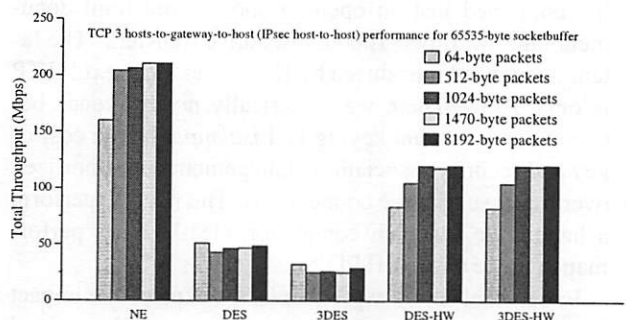


Figure 8: The `ttcp` utility over TCP, for the 3 hosts-to-gateway-to-host network configuration with 65535 bytes of socket buffer. In this case, all 3 hosts on the left form IPsec tunnels to the end host.

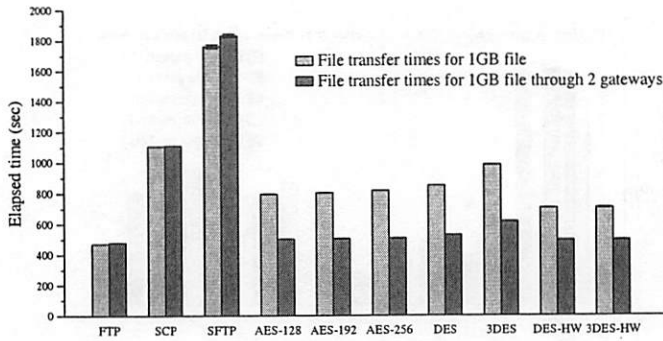


Figure 9: Large file transfer using ftp, scp, sftp, and ftp over IPsec, over two different network topologies. The file is read and stored in the regular Unix FFS. IPsec is not used for the plaintext ftp, scp, and sftp examples, in either setup.

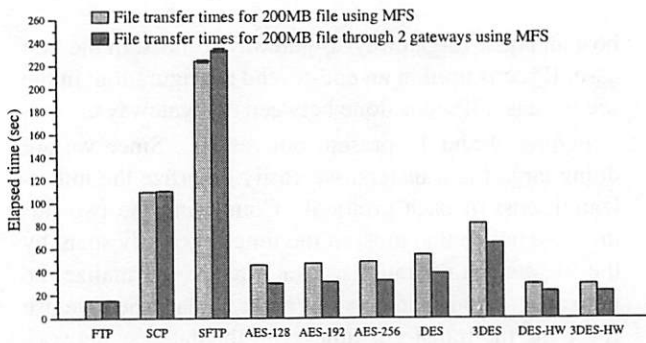


Figure 10: File transfer using ftp, scp, sftp, and ftp over IPsec, over two different network topologies. The file is read and stored in the Unix memory file system (MFS).

used by HTTPS, the network configuration is host-to-host. We used `curl(1)` to transfer a large file from the server to the client. Once again IPsec proves to be a more efficient way of ensuring secure communication.

Figure 13 provides insight on the latency overhead induced by IPsec and HTTPS. We used `curl(1)` to transfer a very small file from the server to the client. The file contained just an opening and closing html document tag. We timed 1000 consecutive transfers. The latency overhead introduced by IPsec over cleartext HTTP is only 10%. There was practically no difference between using manual keying and *isakmpd*, as the cost of key and security association management gets amortized over many successive connections. The need to perform a handshake for each connection clearly hurts performance in the case of HTTPS.

In our final set of experiments, we explore the impact IPsec has on the operation of the system. We selected a CPU-intensive job, *Sieve of Eratosthenes*⁴, which we

⁴Sieve of Eratosthenes is an algorithm for computing prime numbers. We run `primes(6)`, a program that uses this algorithm which is CPU intensive, to emulate a loaded gateway machine

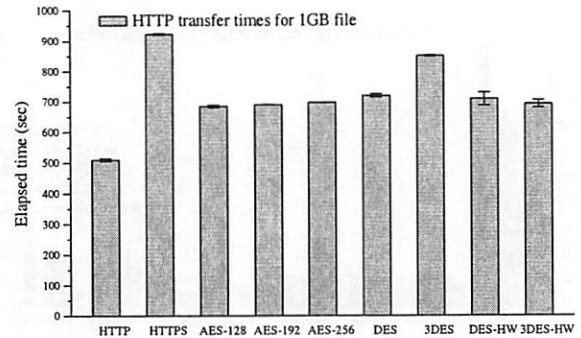


Figure 11: Large file transfer using http, https, and http over IPsec, on a host-to-host network topology. The file is read and stored in the regular Unix FFS.

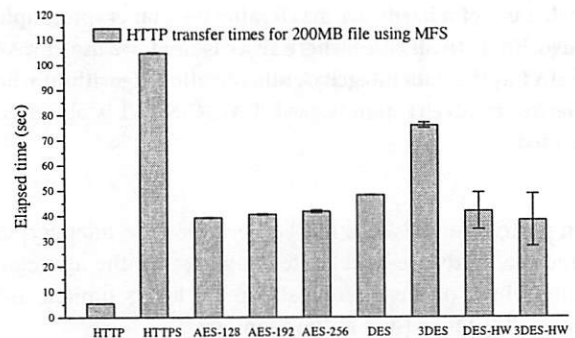


Figure 12: Large file transfer using http, https, and http over IPsec, on a host-to-host network topology. The file is read and stored in the Unix memory file system (MFS).

run while constantly using the network. We tested the impact of a number of protocols to the performance of other jobs (in this case, the sieve) running on the system. In Figure 14, we present the execution times of our CPU intensive job while there is constant background network traffic. To understand the results of Figure 14, one needs to understand how the BSD scheduler works. In BSD, CPU intensive jobs that take up all their quanta have their priority lowered by the operating system. When executing the sieve while using ftp, the sieve program gets its priority lowered and therefore ends up taking more time to finish. In the case where it is run with `scp(1)` or `sftp(1)`, which are themselves CPU intensive because of the crypto operations, the sieve finished faster. When the sieve is run with IPsec traffic, the crypto operations are performed by the kernel and therefore the sieve gets fewer CPU cycles. With hardware cryptographic support, the kernel takes up less CPU which leaves more cycles for the sieve. In the case of HTTPS background network traffic, the CPU cycles spent in crypto processing were not enough to affect the priority of the sieve.

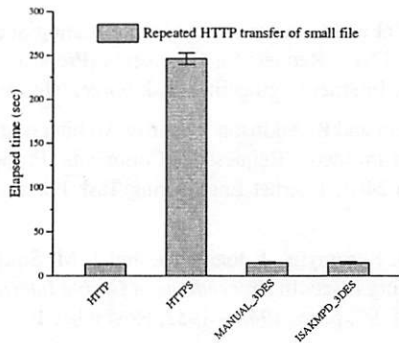


Figure 13: Small file transfer using http, https, and http over IPsec (using manual and automatic keying via *isakmpd*), on a host-to-host network topology. We timed 1000 transfers of the file. The 3DES algorithm was used for encryption.

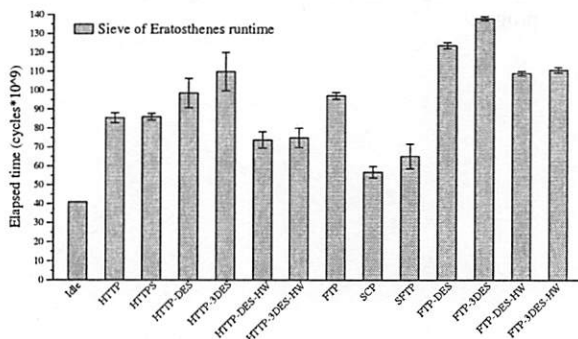


Figure 14: IPsec introduced overhead on the normal performance of a system. Impact on the execution time of CPU intensive job (sieve) on a system that uses IPsec.

3.3 Macro-benchmark Results

All the experiments we run so far were designed to explore specific aspects of the security protocols, under a variety of configurations. In this section we present benchmarks that reflect a more realistic use of these protocols.

For our first macro-benchmark, we created a local mirror of the `www.openbsd.com` site, 728 files and a total of 21882048 bytes, to a server machine. We then used `wget` (1) from a client machine to transfer the whole tree hierarchy over the Intel PRO/1000F network adapters. We used `wget` (1) instead of `curl` (1) because of its support for recursive web transfers. Four different ciphers/modes were used for HTTPS. The HTTPS tests used server certificates. The IPsec tests were conducted using manual keying with DES, 3DES, AES and hardware accelerated DES and 3DES. Finally, for completeness, we also included ephemeral Diffie-Hellman results for HTTPS. We present the results in Figure 15.

Our second macro-benchmark is the compilation of

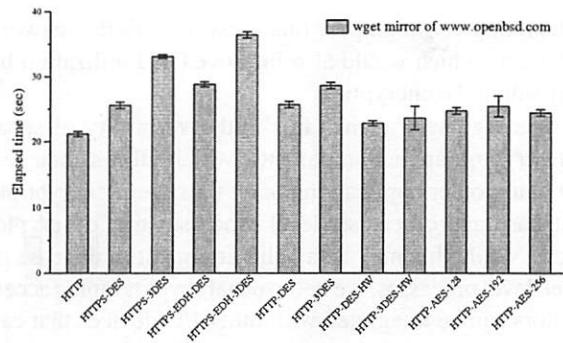


Figure 15: Mass transfer of a web tree hierarchy using `wget`.

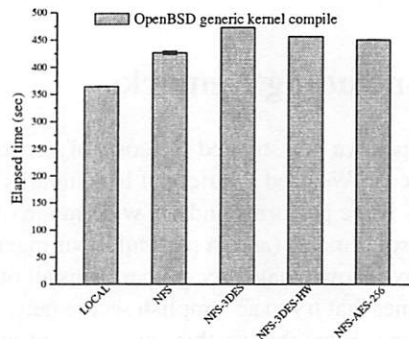


Figure 16: Compilation of the OpenBSD kernel over NFS, with and without use of IPsec.

the OpenBSD source over NFS (see Figure 16). We present results for 3DES with and without hardware support, as well as AES. As expected, using hardware support for the encryption is particularly useful when the system is burdened with intensive CPU and filesystem loads.

4 Discussion

One lesson that can be drawn from our experiments is that the current generation of hardware cryptographic accelerators is not sufficient to support ubiquitous use of encryption. Figure 1 points to one problem: the nominal performance of crypto cards is only achieved for large buffer/packet sizes. Since a large percentage (up to 40%) of the packets in a TCP bulk-transfer is 40 bytes, we can see that much of the benefit of such hardware is lost: the cost of card and DMA initialization, PCI transfers, and interrupt handling is roughly comparable to the cost of pure-software encryption, especially as processor speeds increase. This observation suggests that one possible solution is a hybrid approach, where the system uses software encryption for small packets, and hardware encryption for large ones. Another possible solution could be

integrating cryptographic functionality with the network interface, which would also improve CPU utilization by offloading the encryption.

One argument against this is the versatility of separate cryptographic components, which allows their use by many other applications (*e.g.*, filesystem encryption, database and other user-level processes that do crypto, *etc.*). While this may be a valid argument in the case of user-level processes, we believe that cryptographic accelerators can be integrated with other I/O devices that can use them more efficiently (in particular, disk and tape controllers). The declining cost of high-performance cryptographic chips makes this a viable alternative to dedicated processors.

5 Concluding Remarks

In this paper, we investigated the costs of network security protocols. We used a variety of benchmarks to determine how IPsec performs under a wide range of scenarios. Our experiments (and in particular our macrobenchmarks) have shown that IPsec outperforms all other popular schemes that try to accomplish secure network communications. Even though this safety comes at a price, which is present no matter which protocol one uses, it is possible to get enough performance for practical use by using dedicated cryptographic hardware. This price may easily be acceptable for many applications and environments, given the remarkable flexibility and transparency offered by IPsec.

References

- [1] TTCP: a test of TCP and UDP Performance. USNA, 1984.
- [2] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Comments (Proposed Standard) 2246, Internet Engineering Task Force, January 1999.
- [3] Niklas Hallqvist and Angelos D. Keromytis. Implementing Internet Key Exchange (IKE). In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, pages 201–214, June 2000.
- [4] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.
- [5] John Ioannidis and Matt Blaze. The Architecture and Implementation of Network-Layer Security Under Unix. In *Fourth Usenix Security Symposium Proceedings*. USENIX, October 1993.
- [6] S. Kent and R. Atkinson. IP Authentication Header. Request for Comments (Proposed Standard) 2402, Internet Engineering Task Force, November 1998.
- [7] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). Request for Comments (Proposed Standard) 2406, Internet Engineering Task Force, November 1998.
- [8] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.
- [9] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom) '97*, pages 1948 – 1952, November 1997.
- [10] D. McDonald, C. Metz, and B. Phan. PF.KEY Key Management API, Version 2. Request for Comments (Informational) 2367, Internet Engineering Task Force, July 1998.
- [11] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH Protocol Architecture. Internet Draft, Internet Engineering Task Force, February 1999. Work in progress.

Congestion Control in Linux TCP

Pasi Sarolahti

University of Helsinki, Department of Computer Science

pasi.sarolahti@cs.helsinki.fi

Alexey Kuznetsov

Institute of Nuclear Research at Moscow

kuznet@ms2.inr.ac.ru

Abstract

The TCP protocol is used by the majority of the network applications on the Internet. TCP performance is strongly influenced by its congestion control algorithms that limit the amount of transmitted traffic based on the estimated network capacity and utilization. Because the freely available Linux operating system has gained popularity especially in the network servers, its TCP implementation affects many of the network interactions carried out today. We describe the fundamentals of the Linux TCP design, concentrating on the congestion control algorithms. The Linux TCP implementation supports SACK, TCP timestamps, Explicit Congestion Notification, and techniques to undo congestion window adjustments after incorrect congestion notifications.

In addition to features specified by IETF, Linux has implementation details beyond the specifications aimed to further improve its performance. We discuss these, and finally show the performance effects of Quick acknowledgements, Rate-halving, and the algorithms for correcting incorrect congestion window adjustments by comparing the performance of Linux TCP implementing these features to the performance achieved with an implementation that does not use the algorithms in question.

1 Introduction

The *Transmission Control Protocol (TCP)* [Pos81b, Ste95] has evolved for over 20 years, being the most commonly used transport protocol on the Internet today. An important characteristic feature of TCP are its congestion control algorithms, which are essential for preserving network stability when the network load increases. The TCP congestion control principles require that if the TCP sender detects a packet loss, it should reduce its transmission rate, because the packet was probably dropped by a congested router.

Linux is a freely available Unix-like operating system that has gained popularity in the last years. The Linux source code is publicly available¹, which makes Linux an attractive tool for the computer science researchers in various research areas. Therefore, a large number of people have contributed to Linux development during its lifetime. However, many people find it tedious to study the different aspects of the Linux behavior just by reading the source code. Therefore, in this work we describe the design solutions selected in the TCP implementation of the Linux kernel version 2.4. The Linux TCP implementation contains features that differ from the other TCP implementations used today, and we believe that the protocol designers working with TCP find these features interesting considering their work.

The Internet protocols are standardized by the *Internet Engineering Task Force (IETF)* in documents called *Request For Comments (RFC)*. Currently there are thousands of RFCs, of which tens are related to the TCP protocol. In addition to the mandatory TCP specifications, there are a number of experimental and informational specifications of TCP enhancements for improving the performance under certain conditions, which can be im-

¹The Linux kernel source can be obtained from <http://www.kernel.org/>.

plemented optionally.

Building up a single consistent protocol implementation which conforms to the different RFCs is not a straightforward task. For example, the TCP congestion control specification [APS99] gives a detailed description of the basic congestion control algorithm, making it easier for the implementor to apply it. However, if the TCP implementation supports SACK TCP [MMFR96], it needs to follow congestion control specifications that use a partially different set of concepts and variables than those given in the standard congestion control RFC [FF96, BAF01]. Therefore, strictly following the algorithms used in the specifications makes an implementation unnecessarily complicated, as usually both algorithms need to be included in the TCP implementation at the same time.

In this work we present the approach taken in Linux TCP for implementing the congestion control algorithms. Linux TCP implements many of the RFC specifications in a single congestion control engine, using the common code for supporting both SACK TCP and NewReno TCP without SACK information. In addition, Linux TCP refines many of the specifications in order to improve the TCP efficiency. We describe the Linux-specific protocol enhancements in this paper. Additionally, our goal is to point out the details where Linux TCP behavior differs from the conventional TCP implementations or the RFC specifications.

This paper is organized as follows. In Section 2 we first describe the TCP protocol and its congestion control algorithms in more detail. In Section 3 we introduce the main concepts of the Linux TCP congestion control engine and describe the main algorithms governing the packet retransmission logic. In Section 4 we describe a number of Linux-specific features, for example concerning the retransmission timer calculation. In Section 5 we discuss how Linux TCP conforms to the IETF specifications related to TCP congestion control, and in Section 6 we illustrate the performance effects of selected Linux-specific design solutions. In Section 7 we conclude our work.

2 TCP Basics

We now briefly describe the TCP congestion control algorithms that are referred to throughout this paper. Because the congestion control algorithms play an important role in TCP performance, a number of further

enhancements for the TCP algorithms have been suggested. We describe here the ones considered most important. Finally, we point out a few details considered problematic in the current TCP specifications by IETF as a motivation for the Linux TCP approach.

2.1 Congestion control

The TCP protocol basics are specified in RFC 793 [Pos81b]. In order to avoid the network congestion that became a serious problem as the number of network hosts increased dramatically, the basic algorithms for performing congestion control were given by Jacobson [Jac88]. Later, the congestion control algorithms have been included in the standards track TCP specification by the IETF [APS99].

The TCP sender uses a *congestion window* (*cwnd*) in regulating its transmission rate based on the feedback it gets from the network. The congestion window is the TCP sender's estimate of how much data can be outstanding in the network without packets being lost. After initializing *cwnd* to one or two segments, the TCP sender is allowed to increase the congestion window either according to a *slow start* algorithm, that is, by one segment for each incoming acknowledgement (ACK), or according to *congestion avoidance*, at a rate of one segment in a round-trip time. The *slow start threshold* (*ssthresh*) is used to determine whether to use slow start or congestion avoidance algorithm. The TCP sender starts with the slow start algorithm and moves to congestion avoidance when *cwnd* reaches the *ssthresh*.

The TCP sender detects packet losses from incoming duplicate acknowledgements, which are generated by the receiver when it receives out-of-order segments. After three successive duplicate ACKs, the sender retransmits a segment and sets *ssthresh* to half of the amount of currently outstanding data. *cwnd* is set to the value of *ssthresh* plus three segments, accounting for the segments that have already left the network according to the arrived duplicate ACKs. In effect the sender halves its transmission rate from what it was before the loss event. This is done because the packet loss is taken as an indication of congestion, and the sender needs to reduce its transmission rate to alleviate the network congestion.

The retransmission due to incoming duplicate ACKs is called *fast retransmit*. After fast retransmit the TCP sender follows the *fast recovery* algorithm until all segments in the last window have been acknowledged. During fast recovery the TCP sender maintains the number

of outstanding segments by sending a new segment for each incoming acknowledgement, if the congestion window allows. The TCP congestion control specification temporarily increases the congestion window for each incoming duplicate ACK to allow forward transmission of a segment, and deflates it back to the value at the beginning of the fast recovery when the fast recovery is over.

Two variants of the fast recovery algorithm have been suggested by the IETF. The standard variant exits the fast recovery when the first acknowledgement advancing the window arrives at the sender. However, if there is more than one segment dropped in the same window, the standard fast retransmit does not perform efficiently. Therefore, an alternative called *NewReno* was suggested [FH99] to improve the TCP performance. NewReno TCP exits the fast recovery only after all segments in the last window have been successfully acknowledged.

Retransmissions may also be triggered by the retransmission timer, which expires at the TCP sender when no new data is acknowledged for a while. *Retransmission timeout (RTO)* is taken as a loss indication, and it triggers retransmission of the unacknowledged segments. In addition, when RTO occurs, the sender resets the congestion window to one segment, since the RTO may indicate that the network load has changed dramatically.

The TCP sender estimates packet round-trip times (RTT) and uses the estimator in determining the RTO value. When a segment arrives at the TCP sender, the IETF specifications instruct it to adjust the RTO value as follows [PA00]:

$$\begin{aligned} RTTVAR &<- \frac{3}{4} * RTTVAR + \frac{1}{4} * |SRTT - R| \\ SRTT &<- \frac{7}{8} * SRTT + \frac{1}{8} * R \\ RTO &<- \max(SRTT + 4 * RTTVAR, 1s.) \end{aligned}$$

where R is the measured round-trip time, RTTVAR is variation of the recent round-trip times, and SRTT is the smoothed mean round-trip time based on the recent measurements.

2.2 Enhancements

Recovery from the packet losses is inefficient in the standard TCP because the cumulative acknowledgements allow only one retransmission in a round-trip time. Therefore, *Selective Acknowledgements (SACK)* [MMFR96] were suggested to make it possible for the receiver to

acknowledge scattered blocks of incoming data instead of a single cumulative acknowledgement, allowing the TCP sender to make more than one retransmission in a round-trip time. SACK can be used only if both ends of the TCP connection support it.

Availability of the SACK information allows the TCP sender to perform congestion control more accurately. Instead of temporarily adjusting the congestion window, the sender can keep track of the amount of outstanding data and compare it against the congestion window when deciding whether it can transmit new segments [BAF01]. However, the unacknowledged segments can be treated in different ways when accounting for outstanding data. The conservative approach promoted by IETF is to consider all unacknowledged data to be outstanding in the network. The *Forward Acknowledgements (FACK)* algorithm [MM96] takes a more aggressive approach and considers the unacknowledged holes between the SACK blocks as lost packets. Although this approach often results in better TCP performance than the conservative approach, it is overly aggressive if packets have been reordered in the network, because the holes between SACK blocks do not indicate lost packets in this case.

The SACK blocks can also be used for reporting spurious retransmissions. The *Duplicate-SACK (D-SACK)* enhancement [FMMP00] allows the TCP receiver to report any duplicate segments it gets by using the SACK blocks. Having this information the TCP sender can conclude in certain circumstances whether it has unnecessarily reduced its congestion control parameters, and thus revert the parameters to the values preceding the retransmission. For example, packet reordering is a potential reason for unnecessary retransmissions, because out-of-order segments trigger duplicate ACKs at the receiver.

The *TCP Timestamp option* [BBJ92] was suggested to allow more accurate round-trip time measurements, especially on network paths with high bandwidth-delay product. A timestamp is attached to each TCP segment, which is then echoed back in the acknowledgement for the segment. From the echoed timestamp the TCP sender can measure exact round-trip times for the segments and use the measurement for deriving the retransmission timeout estimator. In addition to the more exact round-trip time measurement, use of TCP timestamps allows algorithms for protecting against old segments from the previous incarnations of the TCP connection.

The timestamp option also allows detection of unnecessary retransmissions. The *Eifel Algorithm* [LK00] sug-

gests that if an acknowledgement for a retransmitted segment echoes a timestamp earlier than the timestamp of the retransmission stored at the sender, the original segment has arrived at the receiver, and the retransmission was unnecessarily made. In such a case, the TCP sender can continue by sending new data and revert the recent changes made to the congestion control parameters.

Instead of inferring congestion from the lost packets, *Explicit Congestion Notification (ECN)* [RFB01] was suggested for routers to explicitly mark packets when they arrive to a congested point in the network. When the TCP sender receives an echoed ECN notification from the receiver, it should reduce its transmission rate to mitigate the congestion in the network. ECN allows the TCP senders to be congestion-aware without necessarily suffering from packet losses.

2.3 Criticism

Some details in IETF specifications are problematic in practice. Although many of the RFCs suggest a general algorithm that could be applied to an implementation, combining the algorithms from several RFCs may be inconvenient. For example, combining the congestion control requirements for SACK TCP and NewReno TCP turns out to be problematic due to different variables and algorithms used in the specifications.

The TCP congestion control specifications artificially increase the congestion window during the fast recovery in order to allow forward transmissions to keep the number of outstanding segments stable. Therefore, the congestion window size does not actually reflect the number of segments allowed to be outstanding during the fast recovery. When fast recovery is over, the congestion window is deflated back to a proper size. This procedure is needed because the congestion window is traditionally evaluated against the difference of the highest data segment transmitted (`SND.NXT`) and the first unacknowledged segment (`SND.UNA`). By taking a more flexible method for evaluating the number of outstanding segments, the congestion window size can be constantly maintained at a proper level corresponding to the network capacity.

Adjusting the congestion window consistently becomes an issue especially when SACK information can be used by the TCP sender. By using the selective acknowledgements, the sender can derive the number of packets with a better accuracy than by just using the cumulative acknowledgements. In order to make a coherent

implementation of the congestion control algorithms, it is desirable to have common variables and routines both for SACK TCP and for the TCP variant to use when the other end does not support SACK.

Finally, the details of the RTO algorithm presented above have been questioned. Since many networks have round-trip delays of a few tens of milliseconds or less, the RTO algorithm details may not have a significant effect on TCP performance, since the minimum RTO value is limited to one second. However, there are high-delay networks for which the effectiveness of the RTO calculation is important. It has been pointed out that the RTO estimator results in overly large values due to the weight given for the variance of the round-trip time, when the round-trip time suddenly drops for some reason. On the other hand, when the congestion window size increases at a steady pace during the slow start, it is possible that the RTO estimator is not increased fast enough due to small variance in the round-trip times. This may result in spurious retransmission timeouts. Alternative RTO estimators, such as the *Eifel Retransmission Timer* [LS00], have been suggested to overcome the potential problems in the standard RTO algorithm. However, although the Eifel Retransmission Timer is efficient in avoiding the problems of the standard RTO algorithm, it introduces a rather complex set of equations compared to the standard RTO. Therefore, evaluating the possible side effects of different network scenarios on Eifel RTT dynamics is difficult.

3 Linux Approach

Although Linux conforms to the TCP congestion control principles, it takes a different approach in carrying out the congestion control. Instead of comparing the congestion window to the difference of `SND.NXT` and `SND.UNA`, the Linux TCP sender determines the number of packets currently outstanding in the network. The Linux TCP sender then compares the number of outstanding segments to the congestion window when making decisions on how much to transmit. Linux tracks the number of outstanding segments in units of full-sized packets, whereas the TCP specifications and some implementations compare `cwnd` to the number of transmitted octets. This results in different behavior if small segments are used: if the implementation uses a byte-based congestion window, it allows several small segments to be injected in the network for each full-sized segment in the congestion window. Linux, on the other hand, allows only one packet to be transmitted for each

segment in the congestion window, regardless of its size. Therefore, Linux is more conservative compared to the byte-based approach when the TCP payload consists of small segments.

The Linux TCP sender uses the same set of concepts and functions for determining the number of outstanding packets with the NewReno recovery and with the two flavors of SACK recovery supported. When the SACK information can be used, the sender can either follow the *Forward Acknowledgements (FACK)* [MM96] approach considering the holes between the SACK blocks as lost segments, or a more conservative approach similar to the ongoing work under IETF [BAF01]. In the latter alternative the unacknowledged segments are considered outstanding in the network. As a basis for all recovery methods the Linux TCP sender uses the equations:

```
left_out <- sacked_out + lost_out
in_flight <- packets_out -
               left_out + retrans_out
```

in defining the number of segments outstanding in the network. In the equation above, `packets_out` is the number of originally transmitted segments above `SND.UNA`, `sacked_out` is the number of segments acknowledged by SACK blocks, `lost_out` is an estimation of the number of segments lost in the network, and `retrans_out` is the number of retransmitted segments. Determining the `lost_out` parameter depends on the selected recovery method. For example, when FACK is in use, all unacknowledged segments between the highest SACK block and the cumulative acknowledgement are counted in `lost_out`. The selected approach makes it easy to add new heuristics for evaluating which segments are lost.

In the absence of SACK information, the Linux TCP sender increases `sacked_out` by one for each incoming duplicate acknowledgement. This is in conformance with the TCP congestion control specification, and the resulting behavior is similar to the *NewReno* algorithm with its forward transmissions. The design chosen in Linux does not require arbitrary adjusting of the congestion window, but `cwnd` holds the valid number of segments allowed to be outstanding in the network throughout the fast recovery.

The counters used for tracking the number of outstanding, acknowledged, lost, or retransmitted packets require additional data structures for supporting them. The Linux sender maintains the state of each outstanding segment in a scoreboard, where it marks the known state

of the segment. The segment can be marked as outstanding, acknowledged, retransmitted, or lost. Combinations of these bits are also possible. For example, a segment can be declared lost and retransmitted, in which case the sender is expecting to get an acknowledgement for the retransmission. Using this information the Linux sender knows which segments need to be retransmitted, and how to adjust the counters used for determining `in_flight` when a new acknowledgement arrives. The scoreboard also plays an important role when determining whether a segment has been incorrectly assumed lost, for example due to packet reordering.

The scoreboard markings and the counters used for determining the `in_flight` variable should be in consistent state at all times. Markings for outstanding, acknowledged and retransmitted segments are straightforward to maintain, but when to place a *lost* mark depends on the recovery method used. With NewReno recovery, the first unacknowledged packet is marked lost when the sender enters the fast recovery. In effect, this corresponds to the fast retransmit of the IETF congestion control specifications. Furthermore, when a partial ACK not acknowledging all the data outstanding at the beginning of the fast recovery arrives, the first unacknowledged segment is marked lost. This results in retransmission of the next unacknowledged segment, as the NewReno specification requires.

When SACK is used, more than one segment can be marked lost at a time. With the conservative approach, the TCP sender does not count the holes between the acknowledged blocks in `lost_out`, but when FACK is enabled, the sender marks the holes between the SACK blocks lost as soon as they appear. The `lost_out` counter is adjusted appropriately.

The Linux TCP sender is governed by a state machine that determines the sender actions when acknowledgements arrive. The states are as follows:

- **Open.** This is the normal state in which the TCP sender follows the fast path of execution optimized for the common case in processing incoming acknowledgements. When an acknowledgement arrives, the sender increases the congestion window according to either slow start or congestion avoidance, depending on whether the congestion window is smaller or larger than the slow start threshold, respectively.
- **Disorder.** When the sender detects duplicate ACKs or selective acknowledgements, it moves to the *Disorder* state. In this state the congestion window

is not adjusted, but each incoming packet triggers transmission of a new segment. Therefore, the TCP sender follows the packet conservation principle [Jac88], which states that a new packet is not sent out until an old packet has left the network. In practice the behavior in this state is similar to the *limited transmit* proposal by IETF [ABF01], which was suggested to allow more efficient recovery by using fast retransmit when congestion window is small, or when a large number of segments are lost in the last window of transmission.

- **CWR.** The TCP sender may receive congestion notifications either by Explicit Congestion Notification, *ICMP source quench* [Pos81a], or from a local device. When receiving a congestion notification, the Linux sender does not reduce the congestion window at once, but by one segment for every second incoming ACK until the window size is halved. When the sender is in process of reducing the congestion window size and it does not have outstanding retransmissions, it is in *CWR (Congestion Window Reduced)* state. CWR state can be interrupted by *Recovery* or *Loss* states described below.
- **Recovery.** After a sufficient amount of successive duplicate ACKs arrive at the sender, it retransmits the first unacknowledged segment and enters the *Recovery* state. By default, the threshold for entering *Recovery* is three successive duplicate ACKs, a value recommended by the TCP congestion control specification. During the *Recovery* state, the congestion window size is reduced by one segment for every second incoming acknowledgement, similar to the *CWR* state. The window reduction ends when the congestion window size is equal to *ssthresh*, i.e. half of the window size when entering the *Recovery* state. The congestion window is not increased during the recovery state, and the sender either retransmits the segments marked lost, or makes forward transmissions on new data according to the packet conservation principle. The sender stays in the *Recovery* state until all of the segments outstanding when the *Recovery* state was entered are successfully acknowledged. After this the sender goes back to the *Open* state. A retransmission timeout can also interrupt the *Recovery* state.
- **Loss.** When an RTO expires, the sender enters the *Loss* state. All outstanding segments are marked lost, and the congestion window is set to one segment, hence the sender starts increasing the congestion window using the slow start algorithm. A major difference between the *Loss* and *Recovery* states is that in the *Loss* state the congestion window is in-

creased after the sender has reset it to one segment, but in the *Recovery* state the congestion window size can only be reduced. The *Loss* state cannot be interrupted by any other state, thus the sender exits to the *Open* state only after all data outstanding when the *Loss* state began have successfully been acknowledged. For example, fast retransmit cannot be triggered during the *Loss* state, which is in conformance with the NewReno specification.

Linux TCP avoids explicit calls to transmit a packet in any of the above mentioned states, for example, regarding the fast retransmit. The current congestion control state determines how the congestion window is adjusted, and whether the sender considers the unacknowledged segments lost. After the TCP sender has processed an incoming acknowledgement according to the state it is in presently, it transmits segments while *in_flight* is smaller than *cwnd*. The sender either retransmits earlier segments marked lost and not yet retransmitted, or new data segments if there are no lost segments waiting for retransmission.

There are occasions where the number of outstanding packets decreases suddenly by several segments. For example, a retransmitted segment and the following forward transmissions can be acknowledged with a single cumulative ACK. These situations would cause bursts of data to be transmitted into the network, unless they are taken into account in the TCP sender implementation. The Linux TCP sender avoids the bursts by limiting the congestion window to allow at most three segments to be transmitted for an incoming ACK. Since burst avoidance may reduce the congestion window size below the slow start threshold, it is possible for the sender to enter slow start after several segments have been acknowledged by a single ACK.

When a TCP connection is established, many of the TCP variables need to be initialized with some fixed values. However, in order to improve the communication efficiency at the beginning of the connection, the Linux TCP sender stores in its destination cache the slow start threshold, the variables used for the RTO estimator, and an estimator measuring the likeliness of reordering after each TCP connection. If another connection is established to the same destination IP address that is found in the cache, the cached values can be used to get adequate initial values for the new TCP connection. If the network conditions between the sender and the receiver change for some reason, the values in the destination cache could get momentarily outdated. However, we consider this a minor disadvantage.

4 Features

We now list selected Linux TCP features that differ from a typical TCP implementation. Linux implements a number of TCP enhancements proposed recently by IETF, such as *Explicit Congestion Notification* [RFB01] and *D-SACK* [FMMP00]. To our knowledge, these features are not yet widely deployed in TCP implementations, but are likely to be in the future because they are promoted by the IETF.

4.1 Retransmission timer calculation

Some TCP implementations use a coarse-grained retransmission timer, having granularities up to 500 ms. The round-trip time samples are often measured once in a round-trip time. In addition, the present retransmission timer specification requires that the RTO timer should not be less than one second. Considering that most of the present networks provide round-trip times of less than 500 ms, studying the feasibility of the traditional retransmission timer algorithm standardized by IETF has not excited much interest.

Linux TCP has a retransmission timer granularity of 10 ms and the sender takes a round-trip time sample for each segment. Therefore it is capable of achieving more accurate estimations for the retransmission timer, if the assumptions in the timer algorithm are correct. Linux TCP deviates from the IETF specification by allowing a minimum limit of 200 ms for the RTO. Because of the finer timer granularity and the smaller minimum limit for the RTO timer, the correctness of the algorithm for determining the RTO is more important than with a coarse-grain timer. The traditional algorithm for retransmission timeout computation has been found to be problematic in some networking environments [LS00]. This is especially true if a fine-grained timer is used and the round-trip time samples are taken for each segment.

In Section 2 we described two problems regarding the standard RTO algorithm. First, when the round-trip time decreases suddenly, RTT variance increases momentarily and causes the RTO value to be overestimated. Second, the RTT variance can decay to a small value when RTT samples are taken for every segment while the window is large. This increases the risk for spurious RTOs that result in unnecessary retransmissions.

The Linux RTO estimator attacks the first problem by giving less weight for the measured mean deviance

(MDEV) when the measured RTT decreases significantly below the smoothed average. The reduced weight given for the MDEV sample is based on the multipliers used in the standard RTO algorithm. First, the MDEV sample is weighed by $\frac{1}{8}$, corresponding to the multiplier used for the recent RTT measurement in the SRTT equation given in Section 2. Second, MDEV is further multiplied by $\frac{1}{4}$ corresponding to the weight of 4 given for the RTTVAR in the standard RTO algorithm. Therefore, choosing the weight of $\frac{1}{32}$ for the current MDEV neutralizes the effect of the sudden change of the measured RTT on the RTO estimator, and assures that RTO holds a steady value when the measured RTT drops suddenly. This avoids the unwanted peak in the RTO estimator value, while maintaining a conservative behavior. If the round-trip times stay at the reduced level for the next measurements, the RTO estimator starts to decrease slowly to a lower value. In summary, the equation for calculating the MDEV is as follows:

```
if (R < SRTT and |SRTT - R| > MDEV) {  
    MDEV <-  $\frac{31}{32} * MDEV + \frac{1}{32} * |SRTT - R|$   
} else {  
    MDEV <-  $\frac{3}{4} * MDEV + \frac{1}{4} * |SRTT - R|$   
}
```

where R is the recent round-trip time measurement, and SRTT is the smoothed average round-trip time. Linux does not directly modify the RTTVAR variable, but makes the adjustments first on the MDEV variable which is used in adjusting the RTTVAR which determines the RTO. The SRTT and RTO estimator variables are set according to the standard specification.

A separate MDEV variable is needed, because the Linux TCP sender allows decreasing the RTTVAR variable only once in a round-trip time. However, RTTVAR is increased immediately when MDEV gives a higher estimate, thus RTTVAR is the maximum of the MDEV estimates during the last round-trip time. The purpose of this solution is to avoid the problem of underestimated RTOs due to low round-trip time variance, which was the second of the problems described earlier.

Linux TCP supports the *TCP Timestamp option* that allows accurate round-trip time measurement also for retransmitted segments, which is not possible without using timestamps. Having a proper algorithm for RTO calculation is even more important with the timestamp option. According to our experiments, the algorithm proposed above gives reasonable RTO estimates also with TCP timestamps, and avoids the pitfalls of the standard algorithm.

The RTO timer is reset every time an acknowledgement advancing the window arrives at the sender. The RTO timer is also reset when the sender enters the *Recovery* state and retransmits the first segment. During the rest of the *Recovery* state the RTO timer is not reset, but a packet is marked lost, if more than RTO's worth of time has passed from the first transmission of the same segment. This allows more efficient retransmission of packets during the *Recovery* state even though the information from acknowledgements is not sufficient enough to declare the packet lost. However, this method can only be used for segments not yet retransmitted.

4.2 Undoing congestion window adjustments

Because the currently used mechanisms on the Internet do not provide explicit loss information to the TCP sender, it needs to speculate when keeping track of which packets are lost in the network. For example, reordering is often a problem for the TCP sender because it cannot distinguish whether the missing ACKs are caused by a packet loss or by a delayed packet that will arrive later. The Linux TCP sender can, however, detect unnecessary congestion window adjustments afterwards, and do the necessary corrections in the congestion control parameters. For this purpose, when entering the *Recovery* or *Loss* states, the Linux TCP sender stores the old *ssthresh* value prior to adjusting it.

A delayed segment can trigger an unnecessary retransmission, either due to spurious retransmission timeout or due to packet reordering. The Linux TCP sender has mainly two methods for detecting afterwards that it unnecessarily retransmitted the segment. Firstly, the receiver can inform by a *Duplicate-SACK (D-SACK)* that the incoming segment was already received. If all segments retransmitted during the last recovery period are acknowledged by D-SACK, the sender knows that the recovery period was unnecessarily triggered. Secondly, the Linux TCP sender can detect unnecessary retransmissions by using the TCP timestamp option attached to each TCP header. When this option is in use, the TCP receiver echoes the timestamp of the segment that triggered the acknowledgement back to the sender, allowing the TCP sender to conclude whether the ACK was triggered by the original or by the retransmitted segment. The *Eifel* algorithm uses a similar method for detecting spurious retransmissions.

When an unnecessary retransmission is detected by using TCP timestamps, the logic for undoing the congestion window adjustments is simple. If the sender is in

the *Loss* state, i.e. it is retransmitting after an RTO which was triggered unnecessarily, the *lost* mark is removed from all segments in the scoreboard, causing the sender to continue with transmitting new data instead of retransmissions. In addition, *cwnd* is set to the maximum of its present value and *ssthresh* * 2, and the *ssthresh* is set to its prior value stored earlier. Since *ssthresh* was set to the half of the number of outstanding segments when the packet loss is detected, the effect is to continue in congestion avoidance at a similar rate as when the *Loss* state was entered.

Unnecessary retransmission can also be detected by the TCP timestamps while the sender is in the *Recovery* state. In this case the *Recovery* state is finished normally, with the exception that the congestion window is increased to the maximum of its present value and *ssthresh* * 2, and *ssthresh* is set to its prior value. In addition, when a partial ACK for the unnecessary retransmission arrives, the sender does not mark the next unacknowledged segment lost, but continues according to present scoreboard markings, possibly transmitting new data.

In order to use D-SACK for undoing the congestion control parameters, the TCP sender tracks the number of retransmissions that have to be declared unnecessary before reverting the congestion control parameters. When the sender detects a D-SACK block, it reduces the number of revertable outstanding retransmissions by one. If the D-SACK blocks eventually acknowledge every retransmission in the last window as unnecessarily made and the retransmission counter falls to zero due to D-SACKs, the sender increases the congestion window and reverts the last modification to *ssthresh* similarly to what was described above.

While handling the unnecessary retransmissions, the Linux TCP sender maintains a metric measuring the observed reordering in the network in variable *reordering*. This variable is also stored in the destination cache after the connection is finished. *reordering* is updated when the Linux sender detects unnecessary retransmission during the *Recovery* state by TCP timestamps or D-SACK, or when an incoming acknowledgement is for an unacknowledged hole in the sequence number space below selectively acknowledged sequence numbers. In these cases *reordering* is set to the number of segments between the highest segment acknowledged and the currently acknowledged segment, in other words, it corresponds to the maximum distance of reordering in segments detected in the network. Additionally, if FACK was in use when reordering was detected, the sender switches to use the conservative variant of

SACK, which is not too aggressive in a network involving reordering.

4.3 Delayed acknowledgements

The TCP specifications state that the TCP receiver should delay the acknowledgements for a maximum time of 500 ms in order to avoid the *Silly Window Syndrome* [Cla82]. The specifications do not mandate any specific delay time, but many implementations use a static delay of 200 ms for this purpose. However, a fixed delay time may not be adequate in all networking environments with different properties. Thus, the Linux TCP receiver adjusts the timer for delaying acknowledgements dynamically to estimate the doubled packet interarrival time, while sending acknowledgements for at least every second incoming segment. A similar approach was also suggested in an early RFC by Clark [Cla82]. However, the maximum delay for sending an acknowledgement is limited to 200 ms.

Using delayed ACKs slows down the TCP sender, because it increases the congestion window size based on the rate of incoming acknowledgements. In order to speed up the transmission in the beginning of the slow start, the Linux TCP receiver refrains from delaying the acknowledgements for the first incoming segments at the beginning of the connection. This is called *quick acknowledgements*.

The number of quick acknowledgements sent by the Linux TCP receiver is at most half of the number of segments required to reach the receiver's advertised window limit. Therefore, using quick acknowledgements does not open the opportunity for the Silly Window Syndrome to occur. In addition, the Linux receiver monitors whether the traffic appears to be bidirectional, in which case it disables the quick acknowledgements mechanism. This is done to avoid transmitting pure acknowledgements unnecessarily when they can be piggybacked with data segments.

4.4 Congestion Window Validation

The Linux sender reduces the congestion window size if it has not been fully used for one RTO estimate's worth of time. This scheme is similar to the *Congestion Window Validation* suggested by the IETF [HPF00]. The motivation for Congestion Window Validation is that if the congestion window is not fully used, the TCP sender

may have an invalid estimate of the present network conditions. Therefore, a network-friendly sender should reduce the congestion window as a precaution.

When the Congestion Window Validation is triggered, the TCP sender decreases the congestion window to half way between the actually used window and the present congestion window. Before doing this, *ssthresh* is set to the maximum of its current value and $\frac{3}{4}$ of the congestion window, as suggested in RFC 2861.

4.5 Explicit Congestion Notification

Linux implements *Explicit Congestion Notification (ECN)* to allow the ECN-capable congested routers to report congestion before dropping packets. A congested router can mark a bit in the IP header, which is then echoed to the TCP sender by an ECN-capable receiver. When the TCP sender gets the congestion signal, it enters the *CWR* state, in which it gradually decreases the congestion window to half of its current size at the rate of one segment for two incoming acknowledgements. Besides making it possible for the TCP sender to avoid some of the congestion losses, ECN is expected to improve the network performance when it is more widely deployed to the Internet routers.

5 Conformance to IETF Specifications

Since Linux combines the features specified in different IETF specifications following certain design principles described earlier, some IETF specifications are not fully implemented according to the algorithms given in the RFCs. Table 1 shows our view of which RFC specifications related to TCP congestion control are implemented in Linux. Some of the features shown in the table can be found in Linux, but they do not fully follow the given specification in all details. These features are marked with an asterisk in the table, and we will explain the differences between Linux and the corresponding RFC in more detail below.

Linux fast recovery does not fully follow the behavior given in RFC 2582. First, the sender adjusts the threshold for triggering fast retransmit dynamically, based on the observed reordering in the network. Therefore, it is possible that the third duplicate ACK does not trigger a fast retransmit in all situations. Second, the Linux sender does not artificially adjust the congestion win-

Table 1: TCP congestion control related IETF specifications implemented in Linux. + = implemented, * = implemented, but details differ from specification.

Specification	Status
RFC 1323 (Perf. Extensions)	+
RFC 2018 (SACK)	+
RFC 2140 (Ctrl block sharing)	+
RFC 2581 (Congestion control)	*
RFC 2582 (NewReno)	*
RFC 2861 (Cwnd validation)	+
RFC 2883 (D-SACK)	+
RFC 2988 (RTO)	*
RFC 3042 (Lim. xmit)	+
RFC 3168 (ECN)	*

dow during fast recovery, but maintains its size while adjusting the `in_flight` estimator based on incoming acknowledgements. The different approach alone would not cause significant effect on TCP performance, but when entering the fast recovery, the Linux sender does not reduce the congestion window size at once, as RFC 2582 suggests. Instead, the sender decreases the congestion window size gradually, by one segment per two incoming acknowledgements, until the congestion window meets half of its original value. This approach was originally suggested by Hoe [Hoe95], and later it was named *Rate-halving* according to an expired Internet Draft by Mathis, et. al. Rate-halving avoids pauses in transmission, but is slightly too aggressive after the congestion notification, until the congestion window has reached a proper size.

As described in Section 4, the round-trip time estimator and RTO calculation in Linux differs from the Proposed Standard specification by the IETF. Linux follows the basic patterns given in RFC 2988, but the implementation differs from the specification in adjusting the `RTTVAR`. A significant difference between RFC 2988 and Linux implementation is that Linux uses the minimum RTO limit of 200 ms instead of 1000 ms given in RFC 2988.

RFC 2018 defines the format and basic usage of the SACK blocks, but does not give detailed specification of the congestion control algorithm that should be used with SACK. Therefore, applying the FACK congestion control algorithm, as Linux does by default, does not violate the current IETF specifications. However, since FACK results in overly aggressive behavior when packets have been reordered in the network, the Linux sender changes from FACK to a more conservative congestion control algorithm when it detects reordering. The IETF

currently has a work in progress draft defining a congestion control algorithm to be used with SACK [BAF01], which is similar to the conservative SACK alternative in Linux. Furthermore, Linux follows the D-SACK basics given in RFC 2883.

Linux implements RFC 1323, which defines the TCP timestamp and window scaling options, and the limited transmit enhancement defined in RFC 3042, which is taken care of by the *Disorder* state of the Linux TCP state machine. However, if the reordering estimator has been increased from the default of three segments, the Linux TCP sender transmits a new segment for each incoming acknowledgement, not only for the two first ACKs. Finally, the Linux destination cache provides functionality similar to the RFC 2140 that proposes Control Block Interdependence between the TCP connections.

6 Performance Issues

Before concluding our work, we illustrate the performance implications of using quick acknowledgements, rate-halving, and congestion window reverting. We do this by disabling these features, and comparing the time-sequence diagrams of a pure Linux TCP implementation and an implementation with the corresponding feature disabled. We use Linux hosts as connection endpoints communicating over a 256 Kbps link with MTU of 1500 bytes. Between the sender and the 256 Kbps link there is a tail-drop router with buffer space for seven packets, connected to the sender with a high-bandwidth link with small latency. The test setup is illustrated in Figure 1. In addition to the low bandwidth, the link between the router and TCP receiver has a fairly high propagation delay of 200 ms. The slow link and the router are emulated using a real-time network emulator [KGM⁺01]. With the network emulator we can control the link and the network parameters and collect statistics and log about the network behavior to help the analysis.

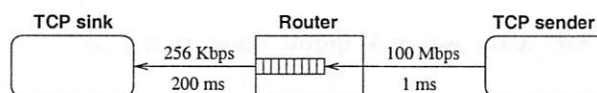


Figure 1: Test setup.

We first illustrate the effect of quick acknowledgements on TCP throughput. Figure 2(a) shows the slow start performance of unmodified Linux implementing quick

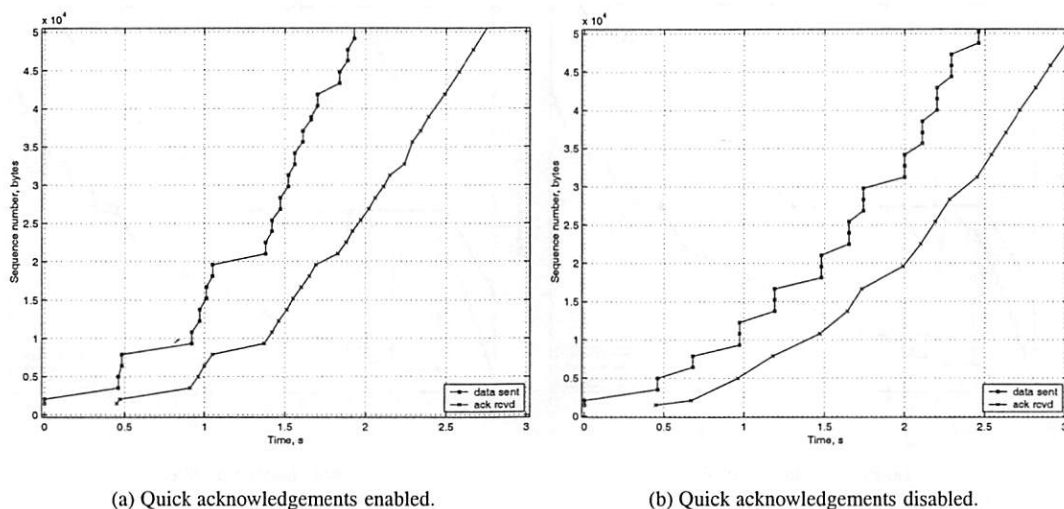


Figure 2: Effect of quick acknowledgements on slow start performance.

acknowledgements, and Figure 2(b) shows the performance of an implementation with the quick acknowledgements mechanism disabled. The latter implementation applies a static delay of 200 ms for every acknowledgement, but transmits an acknowledgement immediately if more than one full-sized segment's worth of unacknowledged data has arrived. One can see that when the link has a high bandwidth-delay product, such as our link does, the benefit of quick acknowledgements is noticeable. The unmodified Linux sender has transmitted 50 KB in 2 seconds, but when the quick acknowledgements are disabled, it takes 2.5 seconds for the sender to transmit 50 KB. In our example, the unmodified Linux receiver with quick acknowledgements enabled sent 109 ACK packets, and the implementation without quick acknowledgements sent 95 ACK packets. Because quick acknowledgements cause more ACKs to be generated in the network than when using the conventional delayed ACKs, the sender's congestion window increases slightly faster. Although this improves the TCP performance, it makes the network slightly more prone to congestion.

Rate-halving is expected to result in a similar average transmission rate as the conventional TCP fast recovery, but it paces the transmission of segments smoothly by making the TCP sender reduce its congestion window steadily instead of making a sudden adjustment. Figure 3(a) illustrates the performance of an unmodified Linux TCP implementing rate-halving, and Figure 3(b) illustrates the performance of an implementation with the conventional fast recovery behavior. These figures

also illustrate the receiver's advertised window (the uppermost line), since it limits the fast recovery in our example.

The scenario is the same in both figures: the router buffer is filled up, and several packets are dropped due to congestion before the feedback of the first packet loss arrives at the sender. The packet losses at the bottleneck link due to initial slow start is called slow start overshooting. The figures show that after 12 seconds both TCP variants have transmitted 160 KB. However, the behavior of the unmodified Linux TCP is different from the TCP with rate-halving disabled. With the conventional fast recovery, the TCP sender stops sending new data until the number of outstanding segments has dropped to half of the original amount, but the sender with the rate-halving algorithm lets the number of outstanding segments reduce steadily, with the rate of one segment for two incoming acknowledgements. Both variants suffer from the advertised window limitation, which does not allow the sender to transmit new data, even though the congestion window would.

Finally, we show how the timestamp-based undoing affects TCP performance. We generated a three-second delay, which is long enough to trigger a retransmission timeout at the TCP sender. Figure 4(a) shows a TCP implementation with the TCP timestamp option enabled, and Figure 4(b) shows the same scenario with timestamps disabled. The acknowledgements arrive at the sender in a burst, because during the delay packets queue up in the emulated link receive buffers and are all re-

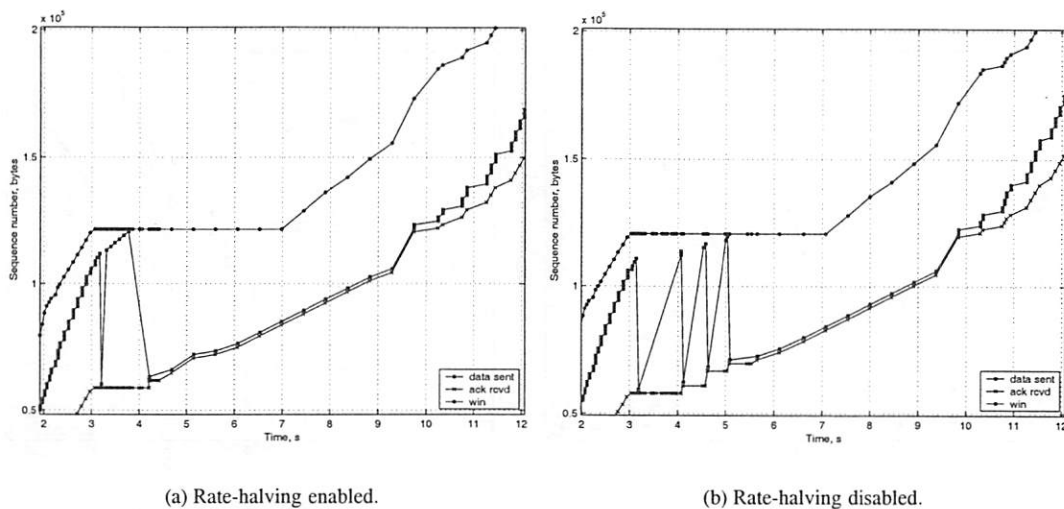


Figure 3: Effect of Rate-halving on TCP performance.

leased when the delay is over².

The timestamps improve the TCP performance considerably, because the TCP sender detects that the acknowledgement following the retransmission was for the original transmission of the segment. Therefore the sender can revert the *ssthresh* to its previous value and increase congestion window. Moreover, the Linux TCP sender avoids unnecessary retransmissions of the segments in the last window. The ACK burst injected by the receiver after the delay causes 19 new segments to be transmitted by the sender within a short time interval. However, the sender follows the slow start correctly as clocked by the incoming acknowledgements, and none of the segments are transmitted unnecessarily.

A conventional TCP sender not implementing congestion window reverting retransmits the last window after the first delayed segment unnecessarily. Not only does this waste the available bandwidth, but the retransmitted segments appearing as out-of-order data at the receiver trigger several duplicate acknowledgments. However, since the TCP sender is still in the *Loss* state, the duplicate ACKs do not cause further retransmissions. One can see that, the conventional TCP sender without timestamps has received acknowledgements for 165 KB of data in the 10 seconds after the transmission begun, while the Linux sender implementing TCP timestamps

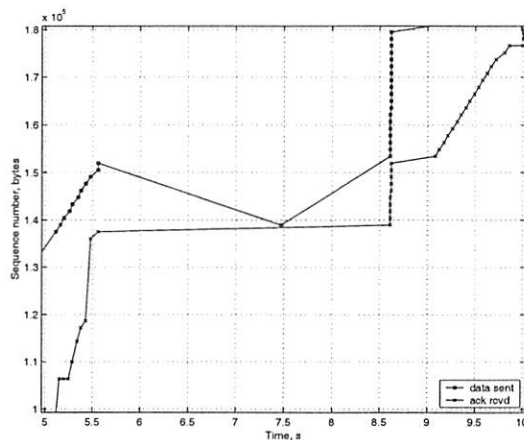
and congestion window reverting has received acknowledgements for 175 KB of data. The Linux TCP sender having TCP timestamps enabled retransmitted 22.6 KB in 16 packets, but the Linux TCP sender without timestamps retransmitted 37.1 KB in 26 packets in the test case transmitting 200 KB. The link scenario was the same in both test runs, having a 3-second delay in the middle of transmission. When TCP timestamps were not used, the TCP sender retransmitted 11 packets unnecessarily.

7 Conclusion

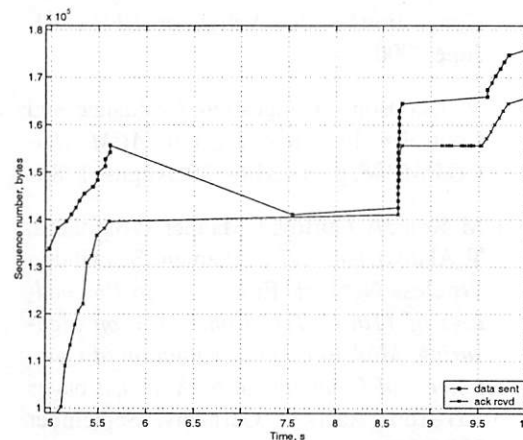
We presented the basic ideas of the Linux TCP implementation, and gave a description of the details that differ from a typical TCP implementation. Linux implements many of the recent TCP enhancements suggested by the IETF, some of which are still at a draft state. Therefore Linux provides a platform for testing the interoperability of the recent enhancements in an actual network. The current design also makes it easy to implement and study alternative congestion control policies.

The Linux TCP behavior is strongly governed by the packet conservation principle and the sender's estimate of which packets are still in the network, which are acknowledged, and which are declared lost. Whether to retransmit or transmit new data depends on the markings made in the TCP scoreboard. In most of the cases none of the requirements given by the IETF are violated, al-

²The delay stands for emulated events on the link layer, for example representing persistent retransmissions of erroneous link layer frames. The link receive buffer holds the successfully received packets until the period of retransmissions is over to be able to deliver them in order for the receiver.



(a) TCP timestamps enabled.



(b) TCP timestamps disabled.

Figure 4: Effect of congestion window undoing on TCP performance.

though in marginal scenarios the detailed behavior may be different from what is given in the IETF specifications. However, the TCP essentials, in particular the congestion control principles and conservation of packets, are maintained in all cases.

The selected approach can also be problematic when implementing some features. Because Linux combines the features in different IETF specifications under the same congestion control engine, an uncareful implementation may break some parts of the retransmission logic. For example, if the balance between congestion window and `in_flight` variable is broken, fast recovery algorithm may not work correctly in all situations.

Acknowledgements

We would like to thank Markku Kojo and Craig Metz for the useful feedback given during preparation of this paper.

References

- [ABF01] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, January 2001.
- [AP99] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, April 1999.
- [BAF01] E. Blanton, M. Allman, and K. Fall. A Conservative SACK-based Loss Recovery Algorithm for TCP. Internet draft "draft-allman-tcp-sack-08.txt", November 2001. Work in progress.
- [BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP Extensions for High Performance. RFC 1323, May 1992.
- [Cla82] D. D. Clark. Window and Acknowledgement Strategy in TCP. RFC 813, July 1982.
- [FF96] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26(3), July 1996.
- [FH99] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999.
- [FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgment (SACK) Option for TCP. RFC 2883, July 2000.
- [Hoe95] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, June 1995.

- [HPF00] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861, June 2000.
- [Jac88] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.
- [KGM⁺01] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a Wireless Network Emulator. In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, pages 151–166, Aachen, Germany, September 2001. VDE Verlag.
- [LK00] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, 30(1), January 2000.
- [LS00] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communication Review*, 30(3), July 2000.
- [MM96] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP Congestion Control. In *Proceedings of ACM SIGCOMM '96*, October 1996.
- [MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, October 1996.
- [PA00] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, November 2000.
- [Pos81a] J. Postel. Internet Control Message Protocol. RFC 792, September 1981.
- [Pos81b] J. Postel. Transmission Control Protocol. RFC 793, September 1981.
- [RFB01] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.
- [Ste95] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1995.

The Future is Coming: Where the X Window System Should Go

Jim Gettys

Cambridge Research Laboratory, Compaq Computer Corporation.

Jim.Gettys@Compaq.com

Abstract

The X Window System was developed as a desktop window system, in a large (for its time) campus scale network environment. In the last few years, it has escaped the desktop and appeared in laptop, handheld and other mobile network devices. X from its inception has been a network transparent window system, and should thrive in this environment. Mobility forces a set of issues to surface that were only partially foreseen in X's design. For one reason or other, the hopes for the design were not entirely realized.

Our original view of X's use included highly mobile individuals (students moving between classes), and a hope, never generally realized for X, was the migration of applications between X servers. Toolkit implementers typically did not understand and share this poorly enunciated vision and were primarily driven by pressing immediate needs, and X's design and implementation made migration or replication difficult to implement as an afterthought. As a result, migration (and replication) was seldom implemented, and early toolkits such as Xt made it very difficult. Emacs is about the only widespread application capable of both migration and replication, and it avoided using any toolkit.

You should be able to travel between work and home or between systems running X at work and retrieve your running applications (with suitable authentication and authorization). You should be able to log out and "park" your applications somewhere until you retrieve them later, either on the same display, or somewhere else. You should be able to migrate your application's display from a handheld to a wall projector (for example, your presentation), and back again. Applications should be able to easily survive the loss of the X server (most commonly caused by the loss of the underlying TCP connection, when running remotely).

There are challenges not fully foreseen: applications must be able to adapt between highly variable display

architectures. Changes to the X infrastructure in recent work make this retrofit into modern toolkits appear feasible, enabling a much more dynamic view of applications. Also, applications must be able to adapt between very different resolution displays (more than an order of magnitude) and differing pointing devices.

I cover the changes and infrastructure required to realize this vision, and hope to demonstrate a compelling part of this vision in action. This vision provides a much more compelling vision of what it means for applications to work in your network. With the advent of high speed metropolitan and wide area networks, and PDA's with high speed wireless networks, this vision will provide a key element of the coming pervasive computing system.

1 Introduction

The X Window System [SG92] architecture lay fallow for at least five years, roughly the period from 1995 to 2000, though many of the extensions defined after the base X11 architecture were questionable at best [Get00]. The good news is that the original X architecture enabled the development of desktop systems such as Gnome [dI99] and KDE [Da99]. But to again match and then exceed other systems such as Microsoft Windows and Apple's Aqua [App] in all areas, serious further work is necessary. This paper attempts to outline areas where base window system work is required to again make X competitive with other current commercial systems, specifically in the current large scale network environment.

I believe that migration and replication of running X applications is key to the near term future environment. A number of issues made replication and/or migration difficult. These include (and some of which are discussed in detail in [GP01]):

- Fonts have been server side objects, and there is

no guarantee the fonts available on one server are available on the other (slightly mitigated by the existence of font servers).

- Graphics operations in X can be made to drawables (windows or off screen memory, called “pixmap”). The X protocol only guarantees 1 bit deep drawables. Displays of different depths might share no other depth. Furthermore, early toolkits typically did not handle image rendering at all, exposing this disparity of displays completely to applications.
- Resource ID’s (for windows, pixmaps, fonts, etc) and Atoms are per server identifiers, and have to be remapped. These were typically exposed directly to applications, though sometimes with some abstraction (for example, partially hidden in a widget data structure).
- Pseudocolor displays again presented toolkits, applications, or pseudoservers major headaches: there would be no guarantee the color resources would be available when needed. It took longer for pseudocolor to become unimportant than expected due to the not fully understood economics that drove displays down market at the same functionality. Thankfully, monochrome (1 bit) and pseudocolor displays are now a thing of the past, for all intents and purposes.

2 Client Side Fonts

While server side fonts clearly provide major headaches for migration and replication (since there is no guarantee that the fonts the application needs are available on the X servers being used), client side fonts are needed on general architectural grounds. This section lays out why.

2.1 Large Scale Distributed Systems Deployment

In large scale commercial systems, applications are often run remotely from the X server to a desktop system or X terminal, and therefore applications won’t be deployable until both ends of the communications have been updated. The X Window System, due to its network transparency, presents the problems of large scale distributed systems. It is infeasible to update all X servers to support new font technology.

The X extension mechanism is at best only a partial solution, since if an application relies on an extension’s existence, it cannot be deployed until the X server has been upgraded. Many older systems can never be updated with new X extensions. Application developers won’t use a new feature if it badly restricts the market for their software.

Applications that will work (even if somewhat slowly) always trump applications that won’t run at all, from the point of view of an ISV (who wants to sell product), or an open source developer who wants their application used widely.

2.2 Historical Observation about Opaque Server side fonts

From the beginning, X’s minimalist font abstraction has presented challenges to applications interested in serious typography. Due to X’s inadequacies, serious applications from the very beginning of X’s history have had to work around X’s fonts, and coordinating X’s fonts with those required for printing has been a 15 year long migraine headache.

Applications need sufficient information for either X’s use, or for printing use (something we ignored in the design of the X protocol). Therefore, to be useful to applications on systems, any access to fonts must be able to access *any* information in the font files, since they will continue to evolve.

There have already been four generations of font formats over X’s history:

- Bitmap 1983-1990
- Speedo 1991
- Type1 1992
- TrueType 1997

OpenType, in the short term future, represents a fifth format. Approximately every 5 years, there is likely to be a new generation of and format for font files, with increasing quality (along some dimension, whether it be typographic, or character set, ...). Any server side font solution, therefore, will always be difficult to deploy in this distributed environment and will end up chasing a moving target.

As each new font technology has deployed, applications have needed to access the new data contained in the font files for the new information. X itself has not evolved to do more than rendering the bits on the screen, and has made it difficult or impossible for applications to share this information.

Over this period, fonts themselves have become much larger. When X was designed, fonts typically had fewer than 128 glyphs (ASCII), and the number of fonts were small, and, since the fonts were bitmap fonts, the metrics were static. The number of round trips to retrieve font metrics were small, and the size of the metrics themselves were small. This situation contrasts greatly with the present.

Many modern fonts have thousands of glyphs and common desktop systems may have hundreds or even thousands of fonts. Searching among these fonts for correct style and code point coverage requires applications to retrieve (typically at startup) very large amounts of metric data, and, to add insult to injury, it is insufficient for most printing applications. Due to this growth in the number and size of fonts, transporting even X's minimal font metrics from the X server to clients now usually swamps transporting the glyph images actually used to the X server from applications. Furthermore, for X to use a scalable font (now the dominant form of fonts used), it must render the font at the specified size, often incurring large startup delays for many applications, before font metrics can be returned to applications, even if few or no glyphs are used (if the font is unsuitable).

The new fontconfig [Pac02] library provides a mechanism for finding the fonts required to fulfill an application's needs, which in concert with Freetype [TT00] allows access directly to the font files the application needs to find the glyphs it needs to render. Round trips to the X server usually dominate performance. Xft [Pac01b] uses the Render extension [Pac01a] when present to cache and render glyphs from the fonts at the X server, avoiding round trips. The Xft2 library now uses only core protocol image transfers if Render is not available, making client-side antialiased fonts (with subpixel decimation) deployable universally. This allows application writers to rely on the new font rendering technology immediately.

Client side fonts therefore have many benefits:

- lower total bandwidth required for modern applications with modern fonts,
- many fewer server round trips,

- incremental behavior: no long startup delays for font rendering,
- as a result, much faster startup time,
- applications can directly access font files to enable printing,
- future font formats can be added without client/server deployment interdependencies,
- an application is guaranteed that the fonts it needs will be usable on any X server in the network, easing migration and replication of applications.

2.3 Remaining work

Some data from modern applications show these benefits [Pac00a] at current screen resolutions (typically 100DPI); more data on a wider selection of applications is needed to confirm this early data.

Worth further investigation is the scaling of glyph data as a function of screen resolution. While sending outlines is certainly possible, this suffers from the deployment problems noted above as this information has changed in each font generation. Naïvely, you might think that the glyph image data would scale as the square of the display resolution, but this is not the case. Even a simple LZW compression algorithm reduces the glyph data to less than linear as a function of resolution; investigation of two dimensional compression techniques is still needed [Clo02].

Further work will be needed therefore to choose a compression technique to ensure that as screen resolution increases, the glyph transport remains efficient. Use of a stripped down PNG [Bea97] like algorithm (to remove redundant header information) is a likely candidate.

Sharing fonts using standard network file systems is much simpler than the current font server situation. This approach can take advantage of the full file sharing / replication / authentication / administration infrastructure being developed for distributed file systems. Using existing infrastructure to enable central administration of fonts is easier than building extensions to the current custom X font service protocol. Applications can get the font information they want and need, and it leverages the full caching/replication protocol work going on in network file system design.

Updating the shared libraries (e.g. Xt, Xaw, Motif, etc) to support AA text is as simple a way to make anti-

aliased text universal than updating the X server and is planned for later in 2002. Modern toolkits such as GTK and Qt have already been updated to support client side fonts, and deployment will be widespread by the end of 2002 on Linux systems.

3 Non-Uniformity of X servers

X servers have varied greatly. The core protocol only guarantees one bit per pixel in addition to the native depth of the screen. This provides a great challenge to applications that might want to migrate or replicate between displays. Further complicating this issue is that historically, toolkits have not provided facilities for rendering of images, relying on applications that need to render images to adapt to the screen correctly (a poor assumption). This means that the visual resources of X servers typically using Xt based toolkits become embedded directly into the applications code, not even abstracted by a toolkit. Pseudocolor displays further complicated the challenge: the color resources required to render an image might or might not be available when needed on an X server

A number of developments over the last four years mitigate this situation.

- The most common X server, XFree86, now uses a new frame buffer package FB, so that all depths are available on all X servers.
- GTK 2.0, and Qt, fully abstract screen resources in ways that completely hide the details of the screen from applications, and provide image abstractions to applications. Applications have to go out of their way to discover information that would make them dependent on the details of rendering of the screen.
- Pseudocolor displays are becoming rare, at long last. So this source of allocation headaches between screens is becoming moot.
- The RandR extension potentially provides the ability to render different depth displays onto a frame buffer, while allowing toolkits to select accelerated visual types and rerender when appropriate. Further work to complete the implementation of RandR is needed, however.

4 Non-Uniformity of Screen size

Screens now vary greatly in size and capability. On the lowest end, the IBM watch provides a 100 by 100 pixel screen. PDA screens are now commonly 320x240 resolution, of small physical size. More conventionally desktop displays are in the 1400x1000 pixel size, with flat panel resolutions increasing. On the high end, within the next two years, some organizations are building large display walls or “caves” with up to 8000x8000 resolution. This very wide disparity of displays provide serious challenges to applications.

Previous attempts to provide migration and replication were often implemented using X servers [GWY94a] that would talk to additional X servers and provide migration and replication services. This model clearly fails across such disparity of screen sizes.

A significant fraction of applications, however, are useful at several display sizes. Personal information management applications are a good example: you want to use these on PDA screens, your desktop, and potentially in a conference room environment on a very large screen. It is unlikely that a single user interface can span this range of display sizes and uses, or deal well with the diversity of pointing devices. GUI builders are a solution for many applications.

The early generation of GUI builders for X were all proprietary, and not universally available, and depended on Motif, which has also not been universally available. Few applications were built using these builders as a result.

The Familiar project has had significant success using Glade [Cha01] and GTK, for example, to provide user interfaces tuned for either portrait or landscape mode on the iPAQ PDA running Familiar Linux. I believe this approach is most suitable to enable many/most applications to be usable across this diversity of screen capabilities. In our examples, the user interface is defined using an XML description built using Glade, and reloaded at run time when the characteristics of the screen changes (in this case, when the screen rotation is changed). I believe this approach is most likely to succeed in avoiding always having to develop applications from scratch for different screen sizes.

Scalable fonts are now widespread, though we must continue to promote their use in existing applications. Some thought, however, needs to be completed on what the abstract size of fonts actually means. X provides the

physical dimensions of the screen, and the number of pixels; it does not, however, provide the typical viewing distance required to allow applications to “do the right thing” with the size of objects to be displayed. In practice, most people’s gut feeling is that a 10 point font is a small, but readable size independent of viewing distance and pixel size; most applications don’t care about the true physical size of the display. We need some convention here to match users’ intuitions.

Notification of the exact characteristics of different X servers will be provided using the RandR extension. A convention to notify applications to migrate or replicate themselves is needed.

5 Resource ID’s

Resource IDs in X are values that only have meaning to a given X server, and are used to identify resources being stored at the X server, such as windows, pixmaps, server side fonts, etc.

Resource IDs are visible in old toolkits, but are generally buried in widgets. In principle, the process of un-mapping/mapping/realizing of widgets provides a mechanism for migration. Unfortunately, almost any sort of drawing beyond basic text was left to applications in Xt; therefore real Xt based applications will likely be difficult to adapt to server migration. Modern toolkits no longer expose resource IDs, visual types or other X resources directly to applications, and since they provide image abstractions, very few applications now have any need to access server dependent X resources directly. This is paying off: patches now exist to allow GTK 2 applications to migrate between dissimilar screens on the same X server; this is much of the work required for migration between X servers. Completion of migration support in GTK and/or Qt is pressing.

6 Connection Failure

TCP connections on wireless networks are more fragile than wired networks, due to signal strength, multipath, and interference, not to mention roaming of the users between networks.

Roaming can be handled via MobileIP [Per95], and X does not pose any special issues here. IPv6, however,

will require some additions in a few areas. While the X core protocol does not have any length dependencies on addresses, some of the ancillary protocols built on top were not built with such foresight and will require some tweaks.

Failure of TCP connections, however, are much more common in the wireless environment. Ideally, applications should be able to survive such losses, and work in toolkits described above should provide most of the required mechanisms. The X library itself, however, needs a small amount of work in this area, to inform toolkits that their connection to the X server has been lost, and allow reclaim of resources. Toolkits could then reconnect themselves to a pseudo X server to allow later retrieval by the user.

There is one possibly significant issue to resolve: in the X protocol, while it is perfectly acceptable (and common) for the X server to send Expose events to clients to request redraw of windows, the core protocol has no such mechanism for pixmaps. In the migration case without connection failure, pixmaps can be retrieved and migrated to the new server. But if the connection has failed, those pixmaps have almost certainly been destroyed, and the client would have to regenerate them. I do not know if this poses an issue to current toolkits or not, since they have taken a higher level of abstraction than Xt based toolkits did, and they may not have problems similar to Xt. Experimentation here is in order, though there have been successful application-specific implementations done in the past [Pac].

7 Replication

Replication poses issues to toolkits, particularly if you would like an application to present different user interfaces on different size screens. This may be hopeless with Xt based toolkits, but may work with modern toolkits.

Pseudo server approaches are likely to work well so long as screen sizes are roughly comparable, given the lack of pseudocolor, the new uniformity of X servers, and the RandR extension. I believe approaches such as HP’s SharedX server [GWY94b] will be useful where toolkit retrofitting is not feasible. Migration will allow sharing via a pseudoserver without a priori arrangement, as applications can migrate to a sharing server. Again, first hand experience is needed.

8 Transparency

Apple's Aqua system for OS X provides full use of alpha blended transparency. The Render extension for X provides part of the equivalent capability for X, by providing alpha blended text and graphics. It does not, however, provide for alpha blended windows, and work here is needed [Pac00b]. That will require significant reimplementation of the device independent part of the X server (which, thankfully, is only a few 10s of thousands of lines of code).

This work is also needed for full implementation of the RandR protocol, to enable rendering to windows that are a different depth than the frame buffer.

9 Authentication and Authorization

Migrating applications presents a serious security problem. You would not want an attacker to be able to hijack your applications, running as you, on your machine. Strong authentication will therefore be required for migration or replication to become widespread.

Support for Kerberos 5 is already in the X library; and SSH provides public key facilities. I do not yet know which may be most appropriate. SSH also provides encryption, which is less urgent (though sometimes vital) than strong authentication. Either or both may be appropriate, and work here is clearly needed, though clearly there are existing facilities than can be exploited to provide the required authentication.

Authorization is another piece of the puzzle, and thought is needed here to understand what is required.

10 Putting the Pieces together

So far, we've discussed the solutions to the individual problems. Here is a sketch of how this might work in practice, to pull the pieces together.

1. A user interface of some sort (whether it be via accelerometer, as discussed in the RandR paper, or something more conventional), would connect to the X server where the application is running, authenticated as you.

2. This application would send a message to the appropriate client(s) to migrate to a destination.
3. The application's toolkit would receive this message, and authenticate that it is from the user.
4. The toolkit opens a new connection to the new X server. If it fails, it would indicate so via some sort of ICCCM message to the user interface, to provide feedback to the user the migration was not possible.
5. The toolkit copies any pixmaps it needs from the original X server to the new one, or creates and rerenders those pixmaps from data it has on hand (probably better, if at all possible). Since all depths are available, the toolkit is guaranteed to be able to copy the bits if needed. This copy phase should occur first, to minimize the likelihood of failure due to inadequate server resources.
6. The toolkit unrealizes itself from the original screen, and realizes itself on the destination screen. If the size of the destination screen is significantly different than the origin screen, the toolkit might reload its user interface definition at this time to provide a more usable user interface. The RandR extension can be used both to provide information on which visual types have acceleration (and a clever toolkit might reconfigure itself to use one of those accelerated visuals). RandR's full implementation also enables rendering to a depth screen different than the actual framebuffer, if the toolkit cannot adapt.
7. At completion, the toolkit would signal its completion on the origin server of the migration it has migrated successfully. A similar sequence should work for connection failures, taking the loss of the TCP connection as a signal the application should attempt to migrate to a home server for later retrieval. Clearly some timeout should be imposed to reap applications after frequent failures. As noted above, there may be some unforeseen problems found in handling connection failure.

11 Plea for help

There are all sorts of projects of various sizes associated with the vision laid out here. I believe that most or all of this vision is achievable. I would like to take this opportunity to solicit others to help realize this vision, which I believe is compelling and should be a lot of fun.

Acknowledgements

The author would like to thank Keith Packard for conspiring on this architecture over the last several years. Additional thanks go to Chris Demetriou for his help in the publication of this manuscript.

References

- [Ano01] Anonymous. Errata: Izzet Agoren's Kernel Corner, May 2001, Mitch Chapman's "Create User Interfaces with Glade" (July 2001). *Linux Journal*, 89:6–6, September 2001. See [Cha01].
- [App] Apple Computer, Inc. AQUA - the mac OS X user experience. <http://www.apple.com/macosx/technologies/aqua.html>.
- [Bea97] T. Boutell et al. RFC2083: PNG (portable network graphics) specification. March 1997. <http://www.ietf.org/rfc/rfc2083.txt>.
- [Cha01] Mitch Chapman. Create user interfaces with Glade. *Linux Journal*, 87:88, 90–92, 94, July 2001. See erratum [Ano01].
- [Clo02] James H. Cloos, Jr. Glyph image compression techniques. Technical report, XFree86, 2002.
- [Dal99] Kalle Dalheimer. KDE: The highway ahead. *Linux Journal*, 58:??–??, February 1999.
- [dI99] Miguel de Icaza. The GNOME project. *Linux Journal*, 58:??–??, February 1999.
- [Get00] James Gettys. Lessons learned about open source. In *Usenix Annual Technical Conference*, 2000. <http://www.usenix.org/publications/library/proceedings/usenix2000/invitedtalks/gettys.html/Talk.htm>.
- [GP01] Jim Gettys and Keith Packard. The X Resize And Rotate Extension - RandR. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, pages 235–244, Boston, MA, June 2001. USENIX.
- [GWY94a] Daniel Garfinkel, Bruce C. Welti, and Thomas W. Yip. HP SharedX: A tool for real-time collaboration. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 45(2):23–36, April 1994.
- [GWY94b] Daniel Garfinkel, Bruce C. Welti, and Thomas W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *HP Journal*, April 1994.
- [Pac] Keith Packard. NCD's WinCenterPro: Networking NT applications using X. <http://www.xfree86.org/keithp/talks/wincen.html>.
- [Pac00a] Keith Packard. An LBX postmortem. Technical report, XFree86 Core Team and SuSE, Inc, 2000.
- [Pac00b] Keith Packard. Translucent windows in x. In *Fourth Annual Linux Showcase & Conference*, pages 39–46, Atlanta, GA, October 2000. USENIX.
- [Pac01a] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, pages 213–224, Boston, MA, June 2001. USENIX.
- [Pac01b] Keith Packard. The xft font library: Architecture and users guide. In *Conference Proceedings*. XFree86 Technical Conference, November 2001.
- [Pac02] Keith Packard. Fontconfig - a shared font configuration mechanism. In *Conference Proceedings*. GNOME Users And Developer European Conference, April 2002.
- [Per95] Charles Perkins. IP mobility support. Technical Report Internet Draft, IETF Mobile IP Group, January 1995. <ftp://software.watson.ibm.com/pub/mobile-ip/draft-ietf-mobileip-protocol-08.txt>.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.
- [TT00] David Turner and The FreeType Development Team. The design of FreeType 2, 2000. <http://www.freetype.org/freetype2/docs/design/>.

XCL : An Xlib Compatibility Layer For XCB

Jamey Sharp Bart Massey
Computer Science Department
Portland State University
Portland, Oregon USA 97207-0751
{jamey,bart}@cs.pdx.edu

Abstract

The X Window System has provided the standard graphical user interface for UNIX systems for more than 15 years. One result is a large installed base of X applications written in C and C++. In almost all cases, these programs rely on the Xlib library to manage their interactions with the X server. The reference implementation of Xlib is as old as X itself, and has been freely available in source form since its inception: it currently is a part of the XFree86 [xfr] distribution.

Unfortunately, Xlib suffers from a number of implementation issues that have made it unsuitable for some classes of application. Most notably, Xlib is a large body of code. This is of most significance on small platforms such as hand-held computers, where permanent and temporary storage are both limited, but can also have performance disadvantages on any modern architecture due to factors such as cache size. In addition, because of Xlib's monolithic nature, it is difficult to maintain.

The authors' prior work on the X protocol C Binding (XCB) is intended to provide a high-quality but incompatible replacement for Xlib. While XCB is believed to be suitable for most new application and toolkit construction, it is desirable to support the large installed base of legacy code and experience by augmenting XCB with an Xlib-compatible API.

This desire has led to the construction of a new library, the Xlib Compatibility Layer (XCL), that is binary-compatible with frequently-used portions of Xlib while being significantly smaller and easier to maintain. Benefits are demonstrated for both existing and new applications written for Xlib. In particular, the significant share of existing knowledge and written material about Xlib remains applicable to XCL. Also, XCL can significantly ease the migration path from Xlib to XCB.

1 The X Window System

The X Window System [SG86] is the *de facto* standard technology for UNIX applications wishing to provide a graphical user interface. The power and success of the X model is due in no small measure to its separation of hardware control from application logic with a stable, published client-server network protocol. In this model, the hardware controller is considered the server, and individual applications and other components of a complete desktop environment are clients.

Development of X began in 1984, and it has become a mature and stable specification that many vendors have implemented for their particular hardware and operating environments. There is now a huge installed base of client applications: X is available for most modern computer systems, and is typically the default on UNIX systems.

To date, most client software for X has been built on top of one or more libraries that hide various details of the protocol, as illustrated in figure 1. Many applications are built using a GUI toolkit, such as Xt [AS90], Qt [Dal01], or GTK+ [Pen99]. These toolkits themselves, however, are almost invariably built on top of Xlib [SGFR92], a library that provides C and C++ language bindings for the X Window System protocol. It is also not uncommon to build applications directly atop Xlib.

2 Xlib and XCB

The authors' recent work has included development of a new X protocol C binding, XCB [MS01], that is intended as a replacement for Xlib in new applications and toolkits. XCB has a number of interesting features, but the XCB API is quite different from the Xlib API: XCB is not intended as a plug-compatible replacement for Xlib in existing applications.

XCB and Xlib are both designed to be C library interfaces to the X Window System network protocol for X

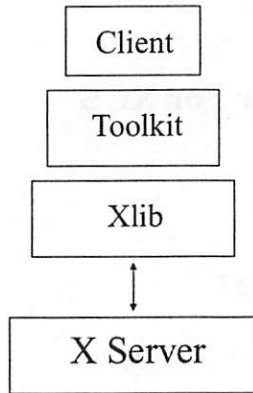


Figure 1: Xlib's role in the X Window System

client applications. However, XCB exchanges Xlib's numerous features for smaller size and enhanced performance. XCB is a lower level API than Xlib: much of the built-in functionality of Xlib (such as caching, display property management, and internationalization) is expected to instead be implemented as separate libraries above XCB, with XCB handling only interaction with the X server at the protocol level.

Some of the Xlib features omitted from XCB are important for getting reasonable performance or proper behavior from an X application. Nonetheless, we believe that these features do not belong in the core protocol library, and should instead be built on top of XCB.

Some of the differences between Xlib and XCB are worthy of detailed consideration.

2.1 Code Generation

The reference implementation of Xlib's core X protocol support alone consists of around 400 files comprising about 100,000 lines of hand-written code. Though the X protocol has a well designed mechanism for protocol extension, and the XFree86 X server allows for reasonably straightforward server-side implementation of these extensions, Xlib has not made the client-side task particularly easy.

In contrast, XCB provides a domain-specific language for specification of the binary encoding of the core protocol and of the majority of extensions. Automated tools translate these protocol descriptions into C implementations. The domain-specific language has significant advantages for maintenance, as well as for implementation of new features and extensions in XCB.

Protocol descriptions may be easily verified against the published specifications for the core X protocol and extensions. Experience with XCB has shown that a brief

inspection of the XCB protocol description for a broken request can quickly lead to a correct fix. This benefit is due to the language making the structure of a request clear, while hiding implementation details in a single location in the source: the protocol-to-C translator.

This encapsulation has also made possible the implementation of a number of useful features, including request marshaling (described next) and tracing of requests and events. Without a common nexus for implementation of these features, the effort required to implement them would have been prohibitive.

2.2 Request Marshaling

For requests that pass to the server a list of independent items, such as a collection of arbitrary line segments, Xlib provides a form of transparent batching known as request marshaling. Marshaling allows client applications to draw individual lines, for example, from different sections of code in rapid succession, without incurring overhead from many similar request headers being sent to the server with small payloads.

Most requests are unsuitable for marshaling, for any of several reasons. If the request is expected to return a reply, combining requests would cause too few replies to be generated. Many requests send only a fixed-length data section with their request header: in these cases, there is generally no room to place additional data. In some cases, requests are not idempotent, so that combining requests would result in different behavior than sending the requests individually. `SetClipRectangles` is one example of this case. (In this paper, we will refer to X protocol requests by the name given to them in the X protocol manual. We will refer to Xlib or XCB functions by their function name: for example, `XSetClipRectangles`.)

XCB also supports marshaling, but treats it as a request attribute that may be specified in the description of the binary encoding of the X protocol. Because of this design, marshaling in XCB is generically supported anywhere in the core protocol or in any extension, although it should only be used if the conditions for correctness given above are satisfied. In the reference implementation of Xlib, marshaling is only used in the places where it is most likely to help: for instance, it will marshal multiple calls to `XDrawLine` but not calls to `XDrawLines`. While this is a wise optimization of programmer time when each function is hand-coded, XCB's use of automated code generation techniques allows it to be more thorough.

2.3 Latency Hiding and Caching

XCB has an easy-to-use latency hiding mechanism intended to allow client applications to get useful work done while waiting for information to return from the server. In XCB, this is accomplished by returning only a placeholder for an expected reply to a server request. Acquiring the placeholder takes no more time than would a request that does not expect a reply. The application can then convert this placeholder into the actual reply data at any time. When the actual data is requested the requesting thread may or may not block. Obtaining the reply data represented by the placeholder takes a small amount of time in most cases: it is quite likely that a request's reply data will already be available by the time the application asks for it.

The reference implementation of Xlib has a mechanism for the same purpose, called an *async-handler*, based on a callback model where one registered function after another is called until one of the callbacks claims responsibility for the data. However, use of Xlib's mechanism is considerably more complex.

Xlib also does extensive client-side caching. Implementation of various caches built on top of XCB is planned as future work, but XCB does no caching. The assumption is that caching can be better managed by a higher layer, for example a toolkit (such as Xt) or a convenience library. In some instances, caching is actually undesirable, either because memory is scarce or because an application needs to know what requests it is generating.

2.4 Thread Safety

Following the successful application of threads in Java's Swing GUI framework, XCB has been designed from the start with the needs of multi-threaded applications in mind, while still supporting single-threaded client applications. While Xlib was designed to support threaded applications, and while that support is not unusable, there are known race conditions that cannot be eliminated without changing the Xlib interface. In particular, if an application needs the protocol sequence number of the request it is making, Xlib forces it to make the request and, without any lock held, query the last sequence number sent. If another thread issues a request between these two events, the first thread will obtain the wrong sequence number. XCB simply returns the sequence number from every request.

2.5 Interfaces

Xlib and XCL have different function signatures for each request in the protocol. In some cases, such as

Xlib's `XCreateSimpleWindow`, the differences are substantial. Still, in many cases these are simple reorderings of the same parameters, since both libraries provide APIs for the same X protocol. The ordering of fields in the protocol specification is optimized primarily for packing efficiency. Xlib uses a parameter ordering based on the parameter ordering of the reference library from version 10 of the X specification. Since X10 compatibility is no longer an important issue, XCB attempts to re-use the existing X11 protocol documentation by keeping its parameter order identical to the order of the fields in the binary protocol specification.

2.6 Feature Comparison

Thanks to the modular design of XCB, applications need only link against the small quantity of code that they actually need. In contrast, Xlib is a monolithic library supporting a variety of disjoint feature sets. Some features are not directly related to the core protocol:

- A color management system consists of mechanisms that enable colors to be displayed consistently across a variety of hardware. Within Xlib is Xcms [SGFR92, Ber95], a complex set of functions for manipulating colors that includes support for device-independent color-spaces. This feature, though a good idea, is layered atop the RGB-based device-dependent color descriptions of the X protocol. While this functionality was included in Xlib in anticipation of its widespread adoption, it has in fact been little-used by applications, and does not appear in XCB. There exist stand-alone color-management libraries [lcm, Gil99] that do not depend on X at all. New code probably should use one of these libraries for color management.
- Xlib has extensive support for internationalization. In particular, it has functions for working with strings of 16-bit characters (such as `XwcDrawText`) and for interaction with input methods. This functionality is not intrinsic to the X Window System, but is conveniently separable. Only the internationalization support inherent in the core protocol is included in XCB. Xlib's internationalization support is becoming less relevant to modern X applications as libraries such as Xft [Pac01b] replace the core protocol font support in order to add anti-aliased fonts and other features unavailable in Xlib.
- Xlib has functions for performing many different kinds of searches on its event queue, including search for particular types of events and on particular windows. XCB provides only a generic queue

traversal, allowing layers closer to the application to provide implementations of predicates that select events.

- Xlib has convenience functions that build on the X protocol's notion of window properties to provide a *resource database*, a sophisticated configuration mechanism for X applications. Since the idea of a resource database is not inherent in the protocol, it is not supported at all by XCB, though the property primitives it relies on are. It would be straightforward to build a compatible resource database atop XCB using window properties.

3 XCL: Xlib Compatibility Layer

XCB has significant advantages over Xlib in some environments, especially small platforms such as handheld computers. However, there are more than 15 years' worth of applications and toolkits built using Xlib. For most existing software, the benefits obtainable by using XCB are outweighed by the effort required to port to it. As an alternative, the XCB design allows it to be used as a replacement "lower layer" for Xlib, efficiently handling X protocol interactions for existing software.

The Xlib Compatibility Layer (XCL) library is an attempt to provide an Xlib-compatible API atop XCB. While Xlib does attempt to layer portions of its implementation, the division between upper and lower layers in Xlib is only visible upon careful examination of the library. Identifying this split is key to providing an Xlib-compatible C library implementing the most commonly used portions of Xlib as a layer on top of XCB, as illustrated in figure 2.

3.1 Xlib Coverage

XCL provides a significant fraction of the functionality of Xlib. This includes adapters from Xlib's protocol request functions to XCB's, implementation of specific predicates for searching XCB's event queue, and a re-implementation of the resource database.

Xlib's interfaces for text internationalization are not provided by XCL. A significant portion of the code size of Xlib is dedicated to translating text strings between various character sets and encodings: XCL deals strictly with glyph rendering using the encoding-neutral interfaces of Xlib. Currently, few Xlib applications and toolkits actually use the internationalized interfaces. Given the decreasing relevance of Xlib's font and text

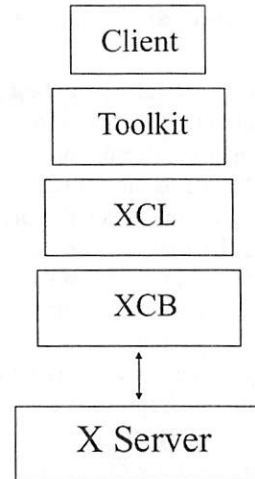


Figure 2: XCL's role in the X Window System

drawing support and the predominance of ASCII encoded text, applications are not expected to need international text rendering functionality from XCL in the future.

Xlib color management is not supported by XCL. The Xlib implementation of this function is by way of integrating an existing color management library: it would be difficult to either duplicate or import this functionality in XCL. In addition, as discussed earlier, few existing applications use the Xlib color management. Many applications use the non-uniform RGB color space provided by Xlib by default. Most applications that do require more sophisticated color management obtain this functionality through toolkit APIs or third-party libraries.

XCL imports some of the caching implementation of Xlib, for example the GCValues cache. In addition, some additional caching is planned, for example an Atom cache. However, substantially less client-side caching is performed by XCL than by the reference Xlib implementation. This decision is a design tradeoff that may need to be revisited as more performance data becomes available.

3.2 Source vs. Binary Compatibility

Many applications and libraries that use Xlib have source code available. In initially specifying requirements for XCL, it was deemed necessary that XCL be at least source-compatible with Xlib, but not necessarily binary-compatible. This weaker requirement leaves open the possibility of sacrificing compatibility of implementation details such as CPP macros and underlying

data structures in order to simplify and ease the implementation.

As it turns out, making XCL largely binary-compatible with Xlib is straightforward. The majority of X applications can use the shared library version of the current XCL implementation as a transparent replacement for the Xlib shared library. Some applications or libraries use obscure features or unpublished interfaces, and may thus need to be recompiled or adapted at the source level.

3.3 Using XCL

XCL is expected to be useful to a variety of different audiences. While each of these target uses is limited, together they cover a broad spectrum of X software development activities. This section enumerates and evaluates some of the expected uses of XCL.

3.3.1 Porting From Xlib to XCB

Applications or toolkits written for Xlib may be ported to XCB in stages using XCL. By simply including `xcl.h` instead of `Xlib.h`, code gains access to the XCB connection structure, and can mix calls to XCB with calls to XCL. Thus software can evolve over time from being entirely Xlib based to being entirely XCB based, claiming the full benefits of XCB.

There are a few cases where the effect of mixing calls to XCL and XCB can be problematic. For example, attempts to manipulate the same graphics context from both XCB and XCL may have unexpected results and lead to client software errors. However, XCL is largely stateless, with most of its state encapsulated by XCB. Most calls to either XCL or XCB should thus produce consistent and predictable effects.

3.3.2 Writing Threaded Code

Xlib is “thread-safe” in the sense that its internal state invariants are protected against simultaneous access by different threads. However, the implementation of thread safety in Xlib is problematic. Internally, a single global lock protects most Xlib state, leading to the possibility of unnecessary interference between unrelated threads and a resulting loss of client efficiency. In addition, the Xlib interface to threading is awkward.

Implementing threaded programs to the XCL interface has a couple of advantages. As noted in the previous section, calls to XCB’s more thread-friendly interfaces

can be used to replace sections of Xlib code of problematic correctness or efficiency. In addition, the finer-grained locking offers the potential for performance enhancement.

3.3.3 Leveraging X Knowledge

There are a wide variety of books available describing programming at the Xlib level, as well as web sites and other documentation. Since the XCL interface is a large proper subset of the published interface of Xlib, all of these resources can still be used by those wishing to learn to write X applications. Those already familiar with Xlib programming can also avoid a steep learning curve while gaining some of the advantages offered by XCL and XCB.

3.3.4 Understanding X Internals

The implementation of XCL may be instructive to those interested in learning about the inner workings of the X Window System. The division of XCL into an upper layer consisting of code borrowed from Xlib and a lower layer replacing some of the more complex and confusing portions of Xlib with calls to XCB may help to illuminate the structure of Xlib and of X protocol bindings in general.

As another example, the transparent translations to the protocol performed by XCB may provide opportunities for better understanding the runtime behavior of Xlib. For example, consider the Inter-Client Communications Conventions Manual (ICCCM) [Ros] that governs interactions between applications and the desktop environment. The ability of XCB to accurately specify and precisely report the ICCCM interactions of an Xlib window manager running atop XCL may be of great benefit in both validating the window manager and increasing developer understanding of ICCCM.

3.3.5 Developing Toolkits

XCB’s latency hiding mechanisms and simple design are expected to be particularly useful to toolkit authors, who can implement optimizations unavailable through Xlib. That expectation applies also to this work. XCL allows toolkit authors to implement XCB-based optimizations in portions of their toolkits needing high performance, while leaving the rest of their Xlib-oriented code base intact. Better yet, toolkits may benefit from further performance gains as XCL and XCB are developed further to support caching and other optimizations.

3.3.6 Accommodating Small Environments

Until XCB becomes widespread on standard systems, applications and toolkits targeted exclusively to XCB will be rare. On the other hand, applications and toolkits targeted to Xlib comprise the bulk of available software. XCL supports these applications by implementing the important portions of Xlib in an extremely lightweight library. The combination of XCL and XCB should provide a platform for legacy X application execution significantly smaller than Xlib. While development should eventually proceed towards new interfaces, support for legacy applications in constrained environments can be a useful feature.

3.4 XCL Influences on XCB

While XCB is largely completed, its development continues in parallel with implementation of XCL. XCB as it currently stands provides a good basis for the XCL implementation. However, in the process of defining XCL, some additional desirable XCB functionality has become apparent.

During the X protocol connection setup phase, a wide variety of per-session server data is sent from the X server to the connecting client. Xlib provides access to this data by way of a plethora of convenience functions. While a clean mechanism for accessing this information has yet to be completed in XCB, a design using accessor functions, including iterators, is currently being implemented. In the meantime, XCB provides direct access to structure accessors for setup data. XCL hides the details of XCB's accessors by copying the data into the XCL `Display` structure in an Xlib-compatible form at connection setup time.

Another aspect of XCB that needs work is its X protocol error handling. In the current implementation, XCB treats X protocol error responses and events similarly, placing error responses in the event queue as though they were events. In addition, XCB does not provide any callback mechanism for error handling, so the only ways to discover that an error has occurred are to process all of the events that precede it in the event queue or to search the event queue using the XCB API. The most significant impact of these XCB design decisions on the design of XCL is that errors often are reported only after dozens of further requests have been processed.

4 XCL Design

XCL copes with two principal APIs. On the client side, it provides the published Xlib API. On the downstream

side, it makes calls to XCB. XCL never communicates directly with the X server: all of its requests and all server responses are routed through XCB.

As discussed previously, XCL's overall architecture is a two-layer affair similar to that of Xlib. Reuse of Xlib code in the XCL upper layer eases verification of the requirement that XCL behave identically to Xlib. In fact, one of the implementation strategies used has been to copy the source for the relevant sections of the reference Xlib implementation, remove sections deemed irrelevant to XCL, and replace them with XCB-based equivalent lower layer code. (Another strategy is discussed in section 4.3.)

It may seem that this implementation approach would make the implementation of XCL relatively trivial, but that is not the case. Different sections of Xlib are coupled together tightly: for example, a significant number of utility and convenience functions contain (sometimes slightly modified) copies of code needed to deliver requests to and accept replies from the X server. XCL makes explicit the idea that a lower layer (XCB) handles communication with the X server. XCL also allows other distinct libraries to handle other jobs, such as caching or color management. In doing so, the interface between the convenience functions and the server has been narrowed and standardized, facilitating future code reuse and maintenance.

XCL's current build process uses the Xlib header files installed on the build machine for their definitions of function prototypes and data structures as an aid to matching the behavior specified for Xlib. This means that as long as the same compiler is used, applications built against these header files are sure to be binary-compatible with XCL, as long as they use only supported interfaces. On the other hand, it means XCL is tied to particular versions of particular implementations of Xlib, a difficulty we encountered early in development. A more detailed analysis of Xlib headers is needed to identify those that should be included in the XCL source.

Another downside to reusing Xlib source in XCL is that Xlib source is not always particularly easy to understand. Some defects in XCL have been caused by misinterpreting the semantics of the Xlib implementation. For example, `XPending` and `XNextEvent` were implemented incorrectly during early development of XCL. The reference implementation of Xlib includes functions `_XReadEvents` and `_XEventsQueued`: both may read events from the X server, but `_XReadEvents` always reads one or more events by blocking until some are available, while `_XEventsQueued` reads zero or more without blocking. As this distinction was missed in a superficial reading, it was assumed that `_XReadEvents` could be used for both jobs. As a re-

sult, it was discovered during testing that test applications would freeze once a window was mapped and the event loop was entered.

Since XCL reuses much of the source of an Xlib implementation, there are a large number of small source files in XCL. Xlib has only one or two functions in most of its source files, so that applications statically linked against Xlib can draw in as little code as possible. While this is a sensible plan for the large and monolithic Xlib, it makes less sense for the small and modular XCL implementation. Future work for XCL includes organizing the source appropriately for its environment.

It is worth noting that the license of the reference Xlib implementation (the MIT license) allows development through code reuse. This licensing policy has made the current XCL implementation possible. Had Xlib been developed under a closed-source or limited-availability license, the task of creating XCL would have been much more daunting. Hopefully, making the source to XCB and XCL freely reusable will allow similar opportunities for future developers.

4.1 XCBConnection and Display

Xlib uses a `Display` data structure to track a single connection to an X server and its state. XCB uses an `XCBConnection` structure for the same purpose. However, these structures have few similarities.

When the XCL `XOpenDisplay` function is called, it in turn calls `XCBConnect` to get an `XCBConnection`. It then sets up a new `Display` structure. In order to support applications that peek into the `Display` structure, XCL copies a number of values from the `XCBConnection` into the `Display`, including the data from connection setup and the file descriptor associated with the server connection. In many applications, this data is part of the required binary interface: although Xlib provides dedicated accessors for this data, the accessors are implemented as CPP macros. It is therefore impossible for XCL to replace the accessors with functions that examine XCB's state directly without breaking binary compatibility. While binary compatibility is not a mandatory requirement of XCL, it is reasonable to provide it when possible: the current architecture permits this.

Early in the development of XCL, an important question arose: how can an `XCBConnection` be associated with a `Display` so that XCL can retrieve the former from the latter? The solution chosen is a standard one: XCL internally manages a structure containing both the `Display` data and a pointer to an `XCBConnection`. The `XCBConnection` pointer is used by internal operations, while the return value of

XCL's `XOpenDisplay` is a pointer to the `Display` data.

4.2 XID Types

The X protocol identifies resources such as windows and fonts using XIDs, small integers that are generated by the client rather than the server. In Xlib-based applications Xlib handles generation of new XIDs as needed. XIDs are declared to be 32-bit integers by Xlib, using C `typedef` statements. This has the significant disadvantage of allowing a variety of type-safety errors: because C types are structurally equivalent, it is easy to use a font XID, for example, where a window XID is required. To remedy this problem, XCB uses a distinct structure type for each kind of resource, allowing the C type system to distinguish between them and ensure that XIDs are used in appropriate contexts.

This causes some difficulty for XCL, however: XCL has to deal with and be able to convert between the two systems of XID types. Conversion from XCB-style to Xlib-style types is easy: the single member of an XCB XID structure is of the same 32-bit integer type as the Xlib XIDs are derived from, and so a structure member access (effectively a type coercion) is all that is needed to perform the conversion. Conversion in the other direction, however, is not so simple.

C does not allow anonymous construction of structure-typed values. To work around this difficulty, XCL provides a number of inline functions that declare a structure of an XCB XID type, initialize it with a value of an Xlib XID type, and return the structure by value. In principal, this could be a source of inefficiency. However, as the bit patterns of the types are identical in both systems, a good optimizing compiler ought to be able to inline and then entirely eliminate the conversion code. In fact, the GNU C compiler does a very good job at this.

4.3 Core Protocol Requests

With XCB handling actual communications tasks, XCL simply needs to manage the data it delivers between XCB and the client application. Figure 3 shows the implementation of `XInternAtom`. The implementation delivers an `InternAtom` request to the X server and extracts the atom XID from the reply data.

Most functions in XCL at this point are at least this simple, if not simpler: when caching, color management, internationalization, and other miscellaneous features of Xlib are removed, the core protocol is the largest piece


```

Atom XInternAtom(Display *dpy,
    const char *name,
    const Bool onlyIfExists)
{
    register xcb_connection_t *c =
        xcb_connection_of_display(dpy);

    Atom atom;
    xcb_intern_atom_reply_t *r;

    if (!name)
        name = "";

    r = xcb_intern_atom_reply(c,
        xcb_intern_atom(c,
            onlyIfExists,
            strlen(name), name),
        0);
    if (!r)
        return None;
    atom = r->atom.xid;
    free(r);
    return (atom);
}

```

Figure 3: XCL implementation of XInternAtom.

that remains. Almost all of the code needed to implement the core protocol is supplied by XCB. Functionality only loosely related to the X protocol should be placed in modules separate from XCL itself, to facilitate maintenance and avoid dragging large amounts of dead code along with every X application.

Some XCL functions are even simpler to implement than XInternAtom, as illustrated by XCreatePixmap in figure 4. As development of XCL has proceeded, common structures became apparent amongst several of the functions that had previously been hand-coded for XCL. In the current implementation, 43 of the roughly 120 requests in the core X protocol are described using a domain-specific language. This language, related to the language used by XCB, describes only the interface to Xlib. A translator to C, implemented in M4 [KR77], reads XCB's interface from XCB's description of the core protocol. Given descriptions of both of the interfaces with which Xlib communicates, the translator can generate the correct code to map parameters between XID type systems and function parameter orderings.

The same benefits expected from XCB's protocol description language (section 2.1) are also expected for XCL's interface description language: maintenance, implementation of new features and extensions, and demonstration of correctness should all be simpler than

```

XCLREQ(CreatePixmap,
    XCLALLOC(Pixmap, pid),
    XCLPARAMS(Drawable drawable,
        unsigned int width,
        unsigned int height,
        unsigned int depth))

```

Figure 4: XCL description of XCreatePixmap.

for hand-coded implementations. The most immediate benefit is that functions generated from this language are known to be implemented to XCB's interface, and can be easily checked against the published Xlib interface specification rather than against the Xlib reference implementation. Future work for XCL thus includes extending this code generation system to implement more requests, including extension requests.

4.4 Protocol Extensions

Since XCB can generate the code for all aspects of the X protocol defined by protocol extensions, implementation of each extension in XCL requires only a proper XCB description of the extension, and any code needed to pre-process requests to or replies from the server.

An example of an extension requiring more than a simple protocol description is the shared memory extension. This extension allows raw data to be transferred through shared memory segments when both the client and the server have access to the same physical memory. XCB can only deal with the X network protocol: shared memory is outside its scope. Thus, an XCL implementation of the shared memory extension would include code to access the shared memory segment, while leaving to XCB the job of exchanging segment identifiers with the server.

4.5 Caching and Latency Hiding

XCL should have as low a latency as reasonably possible from a client call to the desired effect or response. There are two ways to accomplish this goal: hiding the effect of latency from the client application, and removing sources of latency by eliminating high-latency operations.

Caching is an instance of the latter strategy. Little caching is implemented within XCL itself, although graphics context values, for example, are cached using code borrowed from Xlib. Various cache modules will eventually be built directly atop XCB: as these become available XCL will be modified to use them. Until then, most requests through XCL will produce protocol requests to the X server. This can substantially increase

latency in client applications under XCL: fortunately the impact on client applications in current common environments is expected to be minimal.

XCB has latency hiding functionality, and it is desirable to extend this effect to XCL. Unfortunately, Xlib's interface is generally not conducive to this effort. In most cases where a reply is expected, Xlib's interface requires the caller to block until the reply arrives from the X server. However, there are some cases where the interface allows for a potentially large number of requests to be processed in parallel.

XInternAtoms (the plural here is important) is one such case: given a list of atom names to map to atom IDs, it sends requests for all of the atoms to the server, and then waits for each of the replies. The Xlib version of this code, using the async-handler interface discussed in section 2.3, is significantly more complex than the XCL version built on XCB. XCL's XInternAtoms implementation contains considerably less code than Xlib's. (Although, to be fair, this is also because the atom cache has been decoupled from the library and is not yet implemented).

4.5.1 Request Marshaling

Xlib's ability to marshal several independent client drawing requests (such as line drawing requests) into a single protocol request is of minor importance for performance on core protocol requests, at least on modern hardware. However, more recent X extensions, particular the Render extension [Pac01a], depend on request marshaling for performance. In these cases, marshaling opportunities are frequent, and failure to do so may be expensive.

XCB provides request marshaling using a mechanism transparent to clients. In XCB, the ability of a type of request to be marshaled is considered an attribute of the request, and is specified in the same protocol description language as is used to define the binary protocol encoding. Care has been taken to ensure that extensions and the core protocol may be described with the same language, so request marshaling is available to any extension with no more effort on the part of the extension implementor than an extra line of markup.

Because of the transparency of XCB marshaling, the primitive drawing requests in XCL have been trivial to implement, while still performing competitively with their Xlib counterparts. In fact, because only Xlib's XDrawLine (singular) and similar functions marshal in the reference implementation, while XDrawLines (plural) and others do not, some applications could theoretically experience performance gains just by re-linking

with XCL: this could be especially important over low-bandwidth or high-latency links, where marshaling in the core protocol has the most effect. Since marshaling is an intrinsic capability of XCB and not of XCL, all applications built on XCB gain the performance benefits, not just clients built on XCL.

4.6 Threaded Clients

XCL's support for threaded applications is about as complete as the Xlib API will allow. Unfortunately, certain race conditions are still possible, as discussed in section 2.4. Clearly, XCL cannot ensure the correctness of threaded programs written to the Xlib interface. However, it can improve performance for multi-threaded applications.

Xlib uses a single lock to protect the entire contents of its Display data structure, meaning that for most applications all calls into Xlib are serialized. However, there is no reason in principle to disallow multiple threads to access disjoint portions of the internal state simultaneously: while one thread is accessing the event queue, another could be sending a request to the server, and a third could be querying the GC cache.

Currently, XCL allocates a single lock for the entire Display as Xlib does, but for protocol requests that lock is unused. XCB provides a separate thread-safety mechanism for its XCBConnection data that ensures that protocol requests and responses are handled correctly. Any caches implemented atop XCB in the future can (and should) handle their own data structures in a thread-safe fashion.

By leveraging XCB's thread-safety mechanisms, XCL provides fine-grained locking. This may allow multi-threaded applications to take better advantage of the resources of a system than the Xlib implementation would permit.

4.7 Events and Errors

Event and error handling represents a fairly significant portion of the implementation of XCL. Not coincidentally, this portion of the implementation is also one of the least well-specified and understood pieces of Xlib. Several aspects of event and error handling warrant further consideration.

4.7.1 The XCL Event Queue

XCB's event queue is accessed through simple traversal functions. However, this narrow interface is sufficient

for the XCL implementation of Xlib's event search interfaces. Xlib's predicated event retrieval functions may all be implemented as traversals of the event queue, evaluating each event against some arbitrary predicate.

Internally, XCB has a generic linked list implementation that it uses to track several distinct types of information, including a list of replies expected from the server. All users of these internal XCB lists must be able to search for particular items: thus, when the predicated event queue code was added, its implementation added almost no new code to XCB. All of XCB's event queue management implementation combined amounts to only 20 lines of code.

Throughout the design and implementation of XCB and XCL, significant decreases in code size and improvements in maintainability have resulted from the use of modular, layered architecture. Separating list implementations from event queue maintenance and event queue maintenance from XCL predicate implementations provides a nice example of this phenomenon. The modular, layered design provides flexibility, permitting such improvements as reimplementing of internal list interfaces atop new data structures, or pooling of list nodes for reuse. This can be accomplished without modifications to the rest of XCB or XCL, providing improvements transparently to all XCB and XCL client applications.

4.7.2 X Protocol Errors

When XCB reports to XCL that an X protocol error has been received, XCL passes the error off to an error handler as required by the Xlib API. The default error handler displays a (somewhat) human-readable description of what went wrong and terminates the application; however, the handler may be replaced by the client program with an arbitrary client function. This mechanism is awkward to use: as a result, most existing Xlib applications exhibit poor behavior in response to protocol errors. Unfortunately, it is not clear how to address that problem without changing the Xlib error handling API.

4.7.3 XCL and XCB Internal Errors

The reference implementation of Xlib performs various error checks on data coming both from Xlib's callers and from the X server. XCB simply delivers whatever data it is given. The primary benefit of this approach is that XCB may deliver the data faster. An interesting side benefit is that XCB allows for the creation of certain kinds of tools for testing X servers by delivering bad input. Naturally, the trade-off is that XCB does

not particularly help a developer debug a faulty client or server.

In most cases, XCL and XCB have identical failure modes to Xlib, and XCL can return status codes identical to Xlib's when an XCL or XCB error occurs. However, in a small number of instances, XCL can fail in ways that Xlib could not. First, XCL will detect failure in situations that Xlib does not. Second, there may be opportunities for the XCL implementation to fail that were not present in the Xlib implementation (as in the `XDrawString` example below). Unfortunately, Xlib has no particularly well-structured or application-honored mechanism for reporting errors. The current XCL implementation makes reasonable efforts to sensibly handle and report internal errors within the bounds of the Xlib API semantics.

For example, `XDrawString` has to break the string given to it into 254 character chunks, with a two-byte header per chunk. Xlib uses the `Display`'s output buffer directly, while XCL uses `malloc` to create a temporary buffer, subjecting XCL to potential out-of-memory errors. In XCL's current implementation, we call the Xlib-compatible `_XIOError` handler when `malloc` fails.

Despite the fact that even the Xlib implementation of `XDrawString` can fail (if the string provided has a non-positive length), the Xlib implementation always returns success. It would be nice to develop better error handling and reporting mechanisms for these cases within the constraints of the Xlib API.

5 Results

The implementation of XCL is not yet quite complete. However, we have some promising preliminary results.

5.1 RXVT on XCL

The XCL development model includes an initial iteration implementing enough of Xlib to support a single reasonable-sized Xlib application. This is intended to serve as proof of concept, and as validation and evaluation of the design and implementation. The application we chose was the `rxvt` [rxv] terminal emulator, a more modern replacement for `xterm`.

The `rxvt` terminal emulator is related to the standard `xterm` utility in the same way that XCL is related to Xlib. According to the manual, `rxvt` is "intended as an `xterm` replacement for users who do not require features

such as Tektronix 4014 emulation and toolkit-style configurability,” with the benefit that “`rxvt` uses much less swap space.”

This similarity of purpose made `rxvt` an attractive initial target for XCL. In addition, `rxvt` is a reasonably powerful X application that performs very useful work. Finally, `rxvt` exercises a large portion of the core X protocol, including text, rendering and events.

XCL is currently complete enough that `rxvt`, without any source code changes, may be linked against XCL rather than Xlib and will run correctly. In fact, the current stable and development versions of `rxvt` can be compiled against Xlib and then correctly executed against the XCL library binary. The first iteration of XCL development is therefore complete and successful. The remaining work is expected to be largely straightforward, if time-consuming.

5.2 Gnome gw: GDK on XCL

As mentioned in section 3.3.5, one expected use of XCL is with toolkits originally written for Xlib. To test the feasibility of this use, we selected one of the simpler Gnome/GTK+ applications for trials on early versions of XCL. `gw` is a graphical version of the venerable Berkeley “`w`” command used to list the users currently logged into a Unix system.

The `gw` user interface contains a button, a list widget with a scrollbar, and a dockable menu bar. We found that these widgets need only a small subset of Xlib’s functionality. Getting the basic interface to run on XCL was a simple matter: scrollbars scroll, buttons click, menus drop down, and menu bars undock and re-dock. GTK+ and Gnome/GTK+ applications are implemented using a window system independent layer known as GDK. Since the XCL functionality necessary to get `gw` working covers a large portion of the Xlib API subset used by GDK, there is reason to believe that a substantial fraction of the work needed to port GTK+ in its entirety is already completed.

XCL is not perfect yet: some aspects of current `gw` operation atop XCL are visibly wrong, or trigger X protocol errors that shut down the application. `XPutImage` and `XSendEvent`, among others, have been sources of trouble. While fixing these problems may require substantial debugging effort, no major technical barriers are expected.

5.3 Size and Performance

While the implementations of XCL and XCB are still subject to change, it is nonetheless useful to take at least

a first look at the size and performance of the current system. Here are some preliminary metrics:

5.3.1 Size

The size metric used is kilobytes of code in the text section of the compiled object file. The text section needs to be stored both on disk and in memory, both of which are scarce on small systems. In addition, if the entire text section can fit into the L2 cache of the target system (typically between 64kB and 512kB on modern machines), a significant performance improvement might be possible for typical applications making a mix of calls into the library. Including data and BSS size would not significantly impact the reported measurements. All numbers are produced by examining the text sections of either object (.o) files or statically-linked library (.a) files on an Intel x86 architecture machine. Our tests show that use of the optimizer of the compiler is a major factor in the size of both XCB and XCL. The numbers that follow for XCB and XCL are produced by analyzing the output of “`gcc -O2`”: the unoptimized output is nearly 50% larger.

82% of the 79 object files currently comprising XCL have text sections smaller than 512 bytes. The total text section size of XCL is 28kB. XCB adds another 27kB, for a total of 55kB.

Xlib, compiled from 417 source files, has a text section size of 658kB. (According to Jim Gettys [Get01], much of this is data used as lookup tables for internationalization.) The subset of Xlib providing roughly the same functionality as XCL has about 115kB of text. This is more than twice the size of XCL combined with all of XCB. In addition, it would be quite difficult to build a shared library version of Xlib containing just this functionality.

5.3.2 Performance

The XCL version of `rxvt` currently seems to have text display performance comparable to that of the Xlib version. Executing “`cat /usr/share/dict/words`” on a 1GHz Mobile Pentium III running Linux yields runtimes of about 1.3 seconds for `rxvt` 2.7.8 when linked with either Xlib or XCL. Results are similar on a 700MHz Athlon, although XCL consistently exhibits about a 5% speed advantage on this platform for this benchmark.

It is notable that, at about 30,000 scrolled lines per second, `rxvt` is more than acceptably fast for any reasonable use. Indeed, this is the normal case for X appli-

cations on modern hardware: it is extremely unusual to find an application whose performance is limited by interaction with an X server on the same host. In addition, Xlib is believed to provide near-optimal performance in most situations. For these reasons, performance enhancement is not a primary goal of XCL.

6 Related Work

We know of no other efforts to design X protocol client libraries in C. There have been some independent efforts to write such libraries for a variety of other languages including Java [O’N02, Tse01], Common Lisp [SOC⁺89], Smalltalk [Byr], and ML [GR93], as well as more exotic languages such as Python, Erlang, and Ruby. These efforts have concentrated largely on providing natural bindings for their target language, with performance, size, and compatibility with the Xlib API being at most secondary targets.

The Nano-X [Hae01] GUI environment has some interesting parallels to XCL. Nano-X is targeted at lightweight and embedded systems, and provides an API roughly comparable to that of Xlib. However, Nano-X supports a variety of underlying rendering systems, only one of which is the X core protocol. The X protocol support of Nano-X is intended primarily for development and debugging, and is not intended as a principal API for normal application use.

Over the years, Jim Gettys and others have put a significant amount of effort into improvements to the Xlib implementation. Much of this work has been to increase Xlib functionality. More recently, Gettys has put some effort into reducing the size of Xlib for use with Linux on the Compaq iPaq hand-held computer.

The authors’ work on XCL was inspired to some extent by an award-winning program from the 1991 International Obfuscated C Code Contest [NCSB02]. This remarkable 1676 character C program by David Applegate and Guy Jacobson runs Conway’s “Game of Life” cellular automaton on an X root window. Its small size and remarkable performance provided a powerful hint of what is possible with clever coding.

7 Status and Future Work

The XCL implementation is well underway: as noted in the previous section, a useful test application (`rxvt`) links with and runs on XCL. When all Xlib protocol wrapper functions have been implemented in XCL, many more applications are expected to run without modification.

XCL is dependent on the XCB library implementation, which is nearly complete. In its current form, it provides the majority of the functionality needed for the XCL implementation. Some work remains. For example, accessors need to be constructed for variable-element-length lists such as those sent from the X server on connection setup, and XCB error handling needs further exploration.

Allowing a wider variety of applications to run on XCL is the immediate focus as the project continues. One way to quickly support a large number of applications is to support one or more GUI toolkits: preliminary results for GTK+ and Gnome are promising, and we plan to further research this possibility in the near future. As discussed in section 3.3.5, migrating toolkits to XCL may be a good first step in eventually migrating them to XCB.

8 Conclusions

XCL, in conjunction with XCB, represents the first real alternative to Xlib since Xlib’s inception more than 15 years ago. As both are open source, the X development community can examine both for their merits and produce software that is useful for a wide variety of platforms and applications.

Much of the hype surrounding the development of freely-available UNIX software in recent years springs from the idea that open development and the use of freely-available source materials can produce high-quality software products that can then be used to bootstrap future development in this style. XCL provides a nice example of this phenomenon, as well as being a useful tool in its own right. In the immortal words of Hannibal Smith, “I love it when a plan comes together.”

Availability

Current implementations of XCL and XCB are freely available under an MIT-style license at <http://xcb.cs.pdx.edu/>.

Acknowledgements

The authors gratefully acknowledge the advice and assistance of Keith Packard, Jim Gettys, and other X contributors in the design and analysis leading up to XCL. Andy Howe has played a major part in the implementation and testing of XCL. Finally, Chris Demetriou was invaluable throughout the tough task of shepherding this paper.

References

- [AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.
- [Ber95] David T. Berry. Integrating a color management system with a Unix and X11 environment. *The X Resource*, 13(1):179–180, January 1995.
- [Byr] Steve Byrne. *GNU Smalltalk Version 1.1.1 User's Guide*. Web document. URL http://www.cs.utah.edu/dept/old/texinfo/mst/mst_toc.html accessed April 3, 2002 09:12 UTC.
- [Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, 2001.
- [Get01] Jim Gettys, 2001. Personal communication.
- [Gil99] Graeme Gill. Icc file I/O, 1999. Web Document. URL <http://web.access.net.au/argyll/color.html> accessed April 11, 2002 09:25 UTC.
- [GR93] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1, pages 61–80. John Wiley & Sons, 1993.
- [Hae01] Greg Haerr. *Nano-X Reference Manual*, January 2001. Web document. URL <ftp://microwindows.org/pub/microwindows/nano-X-docs.pdf> accessed April 3, 2002 21:00 UTC.
- [KR77] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*. AT&T Bell Laboratories, 1977. Unix Programmer's Manual Volume 2, 7th Edition.
- [lcm] Little CMS. Web Document. URL <http://www.littlecms.com/> accessed April 11, 2002 09:23 UTC.
- [MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.
- [NCSB02] Landon Curt Noll, Simon Cooper, Peter Seebach, and Leonid A. Broukhis. International Obfuscated C Code Contest, 2002. Web document. URL <http://www.ioccc.org/> accessed April 8, 2002 05:58 UTC.
- [O'N02] Eugene O'Neil. XTC: the X Tool Collection, April 2002. Web document. URL <http://www.cs.umb.edu/~eugene/XTC/> accessed April 3, 2002 07:33 UTC.
- [Pac01a] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.
- [Pac01b] Keith Packard. The Xft font library: Architecture and users guide. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.
- [Pen99] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing, 1999.
- [Ros] David Rosenthal. Inter-Client Communication Conventions Manual. In [SGFR92].
- [rxv] RXVT. Web document. URL <http://sourceforge.net/projects/rxvt/> accessed April 8, 2002 5:56 UTC.
- [SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [SGFR92] Robert W. Scheifler, James Gettys, Jim Flowers, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, and XLFD*. Digital Press, third edition, 1992.
- [SOC⁺89] Robert W. Scheifler, LaMott Oren, Keith Cessna, Kerry Kimbrough, Mike Myjak, and Dan Stenger. *CLX Common Lisp X Interface*, 1989.
- [Tse01] Stephen Tse. Escher: Java X11 library, 2001. Web document. URL <http://sourceforge.net/projects/escher> accessed April 3, 2002 20:58 UTC.
- [xfr] The XFree86 project. Web document. URL <http://www.xfree86.org> accessed April 8, 2002 05:54 UTC.

Biglook: a Widget Library for the Scheme Programming Language

Erick Gallesio
Université de Nice – Sophia Antipolis
950 route des Colles, B.P. 145
F-06903 Sophia Antipolis, Cedex
Erick.Gallesio@unice.fr

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Manuel.Serrano@inria.fr

Abstract

Biglook is an Object Oriented Scheme library for constructing GUIs. It uses classes of a CLOS-like object layer to represent widgets and Scheme closures to handle events. Combining functional and object-oriented programming styles yields an original application programming interface that advocates a strict separation between the implementation of the graphical interfaces and the user-associated commands, enabling compact source code.

The Biglook implementation separates the Scheme programming interface and the native back-end. This permits different ports for Biglook. The current version uses GTK+ and Swing graphical toolkits, while the previous release used Tk. It is available at: <http://kaolin.unice.fr/Biglook>.

Introduction

We have studied the problem of constructing GUI in functional languages by designing a widget library for the Scheme programming language, called Biglook. In this paper we focus on how to apply the functional style to GUIs programming.

Biglook's primary use is to implement graphical applications (e.g., xload, editors *à la* Emacs, browser *à la* Netscape, programming tools such as kbrowse, our graphical Scheme code browser). Figure 1 presents two screen

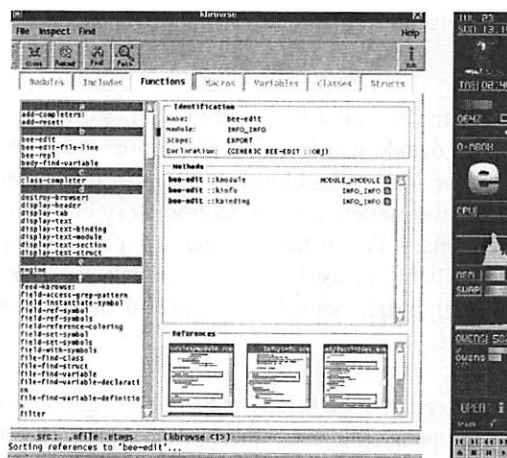


Figure 1: Two Biglook applications: a code browser on the left, “dock” applications on the right

shots of Biglook applications: i) kbrowse on the left and ii) the Biglook dock applications, *à la* NextStep, on the right. These Biglook applications are used on a daily basis.

By contrast to previous work, no attempt has been made to make that library familiar to programmers used to imperative or purely object oriented programming style. On the contrary, our library introduces an original application programming interface (API) that benefits from the high level constructions of the extended Scheme implementation named Bigloo [25] which is open source and freely available since 1992. The main Bigloo component is an optimizing compiler that delivers small and efficient applications for the Unix™ operating system. Bigloo is able to produce native code (via C) and JVM bytecode. Currently Biglook uses GTK+ [20] associated with

the Bigloo C back-end and Swing [29] with the Bigloo JVM back-end. The previous release of Biglook [12] used Tk [19].

Bigloo implements an object layer inspired by CLOS [1]. It is a class-based model where methods override generic functions and are not declared inside classes as in Smalltalk [13], O'Caml [22] or Java [14].

Biglook is implemented as a wrapping layer on top of native widget libraries (that we name henceforth the *back-end*). This software architecture saves the effort of implementing low-level constructions (pixel switching, clipping, event handling and so on) allowing to focus on the Scheme implementation of new features.

When designing the Biglook API, we always had to decide which model to choose: the functional model or the object model. We think that these two models are not contradictory but complementary. For instance, if the widget hierarchy naturally fits a class hierarchy, user call-backs are naturally implemented by the means of Scheme closures.

In Section 1 we briefly present the Bigloo system emphasizing its module system and its object layer. This section is required for readers unfamiliar with the CLOS model and it also serves as an introduction to *virtual slots*. *Virtual slots* are a new Bigloo construction that is required in order to separate the Biglook API made of classes and the native back-end. They are presented in Section 2. In Section 3 we present the Biglook library. We start showing a simple Biglook application and its associated source code. Then, we detail the Biglook programming principles (widgets creation, event handling, etc.) motivating our design orientations by programming language considerations. In this section we are conducted to compare the functional programming style and the object oriented one. In Section 4 we present the Biglook implementation. At last, in Section 5 we present a comparison with related work.

1 Bigloo

Bigloo is an open implementation of the Scheme programming language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme [17]. Bigloo does not implement “all” of Scheme; for example, the execution of tail-recursion may allocate memory. On the other hand, Bigloo implements numerous extensions: support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface, an object layer and a module language. In this Section, we present Bigloo’s modules and its object model; that is, class declarations and generic functions. *Virtual slots*, which are heavily used in Biglook, are presented in Section 2.

1.1 Modules

Bigloo modules have two basic purposes: one is to allow separate compilation and the second is to increase the number of errors that can be detected by the compiler. Bigloo modules are simple and they have been designed with the concern of an easy implementation.

A module is a compilation unit for Bigloo. It is represented by one or more files and has the following syntax:

```
(module module-name
  (import import+)*
  (export export+)*
  (static static+)*
  (library static+)*
)
opt-body
```

Import clauses are used to import bindings in the module. In order to import, one just needs to state the identifier to be imported and its source module. Note that a shorthand exists to import all the bindings of a module.

Export and *static* clauses play a close role. They point out to the compiler that the module implements some bindings and distinguish those that can be used within other modules (they are exported) and those that cannot (they are static). These clauses do not contain identifiers but prototypes. It is then possible to export variables (mu-

table bindings) or functions (read-only bindings). Static clauses are optional (the bindings of a module which are not referenced in a clause are, by default, static).

Library clauses enable programs to use Bigloo libraries. A Bigloo library is merely a collection of pre-compiled modules. Using a library is equivalent to *importing* all the modules composing the library. Detailed information on Bigloo modules may be found in the two papers [25, 26].

1.2 Object layer

In this paper we assume a CLOS-like object model with single inheritance and single dispatch. The object layer implemented in Bigloo is a restricted version of CLOS [1] inspired, to a great extent, by MEROON [21].

1.2.1 Class declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is:

```
(class class-id::super-class-id opt-init opt-slots)
```

A class can inherit from a single super class. Classes with no specified super class inherit from the object class. The type associated with a subclass is a subtype of the type of the super class.

A class may be provided with an initialization function (*opt-init*) that is automatically called each time an instance of *class-id* is created. Initialization functions accept one argument, the created instance.

A slot may be typed (with the annotation *::type-id*) and may have a default value (default option). Here are some possible declarations for the traditional *point* and *point-3d*:

```
(module module-points
  (export (class point
            (point-init)
            (x::double (default 0.0))
            (y::double (default 0.0)))
          (class point-3d::point
            (z::double (default 0.0)))))

(define point-init
  (let ((count 0))
    (lambda (obj::point)
      (set! count (+ 1 count))
      (print "# of points: " count))))
```

1.2.2 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate *cla?*, an allocator *instantiate::cla*, an instance cloner *duplicate::cla*, accessors (e.g., *cla-x* for a slot *x*), modifiers (e.g., *cla-x-set!* for a slot *x*) and an abbreviated special access form, *with-access::cla*, to allow accessing and writing slots simply by using their name. Here is how to allocate and access an instance of *point-3d*:

```
(let ((p (instantiate::point-3d
        (y -3.4)
        (x 1.0))))
  ;; The initialization value of a slot can be omitted
  ;; from the arguments list if it has a default value;
  ;; this is the case for the slot z of class point-3d.
  (with-access::point-3d p (x y z)
    (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

The *instantiate* and *with-access* special forms are implemented by the means of macros that statically resolve the keyword parameters (such as *x*, *y* and *z*). For instance, the above example is expanded into:

```
(let ((p (make-point-3d 1.0 -3.4 0.0)))
  (sqrt (+ (sqr (point-3d-x p))
           (sqr (point-3d-y p))
           (sqr (point-3d-z p)))))
```

As we can see, since macros are expanded at compile-time, there is no run-time penalty associated with keyword parameters.

1.2.3 Generic functions and methods

Generic function declarations are function declarations annotated by the generic keyword. They

can be exported, which means that they can be used from other modules and that methods can be added to those functions from other modules. They can also be static, that is, not accessible from within other modules, which means that no other modules can add methods. The CLOS model fits harmoniously with the traditional functional programming style because a function can be thought as a generic function overridden with exactly one method. The syntax to define a generic function is similar to an ordinary function definition:

```
(define-generic (fun::type arg::class ...) opt-
body)
```

Generic functions must have at least one argument as this will be used to solve the *dynamic dispatch* of methods. This argument is of a type *T* and it is impossible to override generic functions with methods whose first argument is not of a subtype of *T*. Methods are declared by the following syntactic form:

```
(define-method (fun::type arg::class ...) body)
```

Methods override generic function definitions. When a generic function is called, the most specific applicable method, that is the method defined for the closest dynamic type of the instance, is dynamically selected. A method may explicitly invoke the next most specific method overriding the generic function definition for one of its super classes (a class has only one *direct* super class but several *indirect* super classes) by the means of the `(call-next-method)` form. It calls the method that should have been used if the current method had not been defined.

Here is an example of a generic function that illustrates the use of the Bigloo object layer. We are presenting a function that prints the value of the slots of the `point` and `point-3d` instances. This generic function is named `show`:

```
(define-generic (show o::point))
```

Then, the generic function is overridden with a method for classes `point` and `point-3d`.

```
(define-method (show o::point)
  (with-access::point o (x y)
    (print "x=" x)
    (print "y=" y)))

(define-method (show o::point-3d)
  (with-access::point-3d o (z)
    (call-next-method)
    (print "z=" z)))
```

Hereafter is an example of a call to the `show` generic function.

```
(let ((p (instantiate::point-3d
        (x 10)
        (y 20)
        (z 465))))
  (show p))
```

2 Virtual slots

Bigloo supports two kind of instance slots: regular slots that have already been described in Section 1.2.1, and *virtual slots* that enable several views of a single data. As we will see in Section 4.2, *virtual slots* are at the heart of the Bigloo implementation. They are mandatory to present Bigloo to the user as a class based API. In particular, wrapping native widgets (such as GTK+ or Swing) for the Bigloo object model requires virtual slots.

Using virtual slots gives the illusion of accessing the slots of a class instance but instead, Scheme functions are called. As we have seen in Section 1.2.2, the compiler automatically defines *getters* and *setters* that access the various values embedded in the instances regular slots. Accessing virtual slots is syntactically identical to accessing plain slots, but virtual slots differ in the following way:

- their getters and setters are not generated by the compiler. They are defined by the user, in the class definition, using the class slot options: `get` and `set`.
- they are not allocated into memory.

For instance, let us consider a possible rectangle class implementation. An instance of `rect` is characterized by its origin (`x0`, `y0`) and either

its upper right point (x1, y1) or its dimension (width, height). In the following class definition, the width and height slots are virtual.

```
(class rect
  x0 y0 x1 y1
  (width (get (lambda (o)
    (with-access::rect o (x0 x1)
      (- x1 x0))))
    (set (lambda (o v)
      (with-access::rect o (x0 x1)
        (set! x1 (+ x0 v))))))
  (height (get (lambda (o)
    (with-access::rect o (y0 y1)
      (- y1 y0))))
    (set (lambda (o v)
      (with-access::rect o (y0 y1)
        (set! y1 (+ y0 v)))))))
```

Setting the width virtual slot (resp. the height slot) automatically adjusts the x1 value (resp. the y1 value) and *vice versa*. No memory is allocated for width and height, as their *values* are computed each time they are accessed.

3 The Biglook library

Biglook is an Object Oriented Scheme library for constructing GUIs. It offers an extensive set of widgets such as labels, buttons, text editors, gauges, canvases. Most of the functionality it offers are available through classes rather than through *ad-hoc* functions. For instance, instead of having the classical functions `iconify`, `deiconify` and `window-iconified?` for the window widget, Biglook offers the virtual slot `visible` to implement this functionality. Setting the `visible` slot enables iconification/deiconification. Reading the `visible` slot unveils the window iconification state. As we will see, this design choice yields a simpler API and allows usage of introspective techniques for GUIs programming.

In Section 3.1 we first present a small interface and discuss how to create the widgets which compose it in Section 3.2. Section 3.4 describes the notion of *container* widget and placement rules. Finally, in Section 3.5 we show how to make a widget reactive to an external event such as a mouse click.

Throughout this section we justify the choices we have made when designing the Biglook API.

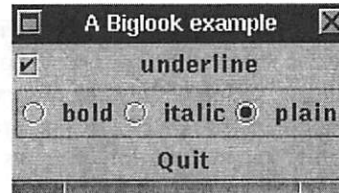


Figure 2: A simple example

Our reflection on how to create a widget or how to handle interfaces events are presented in Sections 3.3 and 3.6.

3.1 A Biglook example

Biglook uses a declarative model for constructing GUIs. This permits a clear separation between the code of the interface and the code of the application. The construction of an interface starts by declaring the various widgets which compose it. Then, the behavior of each widget is specified independently of its creation by associating an action (a Scheme closure) with a widget specific event (key pressed, mouse click, mouse motion, ...).

```
1:(module example (library biglook))
2:
3:(define awin
4:  (instantiate::window
5:    (title "A Biglook example")))
6:(define acheck
7:  (instantiate::check-button
8:    (parent awin)
9:    (text "underline")))
10:(define aradio
11:  (instantiate::radio
12:    (parent awin)
13:    (orientation 'horizontal)
14:    (border-width 2)
15:    (texts '("bold" "italic" "plain"))))
16:(define abutton
17:  (instantiate::button
18:    (parent awin)
19:    (relief 'flat)
20:    (text "Quit")))
```

Figure 3: A simple example, the source code

Figure 2 is a screen shot of a simple Biglook application. It is made of a window (here named "A Biglook example"), a check button (the toggle button `underline`), a radio button group (`bold`, `italic`, `plain`), and a plain button (`Quit`). The

source code of this example is given Figure 3. From that code, we see that in order to access Biglook classes and functions, the program uses a library module clause line 1. This program creates a window (line 4), and three widgets (line 7, line 11 & 17).

In the sections 3.2 and 3.4 we present how to create widgets and how to place them in a window. The Figure 2 interface is inert, that is, no action is associated with the widgets yet. We will see in Section 3.5 how actions can be associated with widgets.

3.2 Widget Creation

The graphical objects (i.e., *widgets*) defined by the Biglook library such as menus, labels or buttons are represented by Bigloo classes. Each class defines a set of slots that implement the configuration of the instances. Consequently, tuning the look of a widget consists in assigning correct values to its slots. The library offers *standard* default values for each widget but these values can of course be changed. Generally the customization is done at widget creation time. For instance, the radio group of Figure 3 will be displayed horizontally and with a border size of 2 pixels (lines 13 and 14). A particular aspect of a widget can be changed by setting a new value to its corresponding slot. For instance, the expression

```
(radio-orientation-set! aradio 'vertical)
```

changes the orientation of the radio group forcing a re-display of the whole window. Of course, the value of this slot can be retrieved by just reading it:

```
(print (radio-orientation aradio))
```

Remember that, as seen in Section 1.2.2, instead of using the functions created by Bigloo that fetch and write the value of a slot, one may write an equivalent program using the Bigloo special form *with-access*:

```
(with-access::radio aradio (orientation)
  (set! orientation 'vertical)
  (print orientation))
```

In the rest of this paper, we will use either forms for accessing class slots.

3.3 Reflection on widget creation

Biglook widget creation supports variable number of arguments and keyword parameters (parameters that can be passed in any order because their actual value is associated with a name). We have found these features very useful in order to enable declarative programming for GUI applications. For instance, a plain Biglook button is characterized by 20 slots. Some of them describe the graphical representation (colors, border sizes, ...), some others describe the internal state of the button (widget parent, associated value, associated text, ...). In general, these numerous slots have default values. When an instance is created, only slots that have no default values must be provided. Slots are initialized with their default value unless a user value is specified. As a consequence, the form that operates widget construction (e.g., class instantiation) must accept a variable number of arguments. Only some slot values must be provided, others are optional. In addition, because widget constructors accept a large number of parameters, it is convenient to *name* them and to be able to pass them in any order. This is made possible in Biglook by the *instantiate::* form. As this form is implemented using macros that are expanded into calls to the class constructors where each declared slot is provided with a value, there is no run-time overhead associated with forms such as *instantiate::*.

Lacking variable number of arguments or keywords disables declarative programming style for GUIs because widgets have to be created and, in a second step, specific attributes have to be provided. Even overloading and class constructors do not help. Let's suppose our window classes implemented in Java AWT [15] or Swing [29]. To enable a full declarative style, we should provide the button class definition with 2^{20} constructors (a constructor for each possible combination of provided slots). Even for much smaller classes, this is impractical because, in general, overloading dispatches on types only and several slots can have the same type. For instance, imagine that we want to change the graphical appearance of our window. Instead of using the smallest area large

enough to display the three widgets, we want to force the width of the window to a specific value. We can turn the definition of Figure 3, line 4 to:

```
(define awin
  (instantiate::window
    (title "A Biglook example")
    (width 300)))
```

If we want to specify both width and height, we can use:

```
(define awin
  (instantiate::window
    (title "A Biglook example")
    (width 300)
    (height 200)))
```

Languages relying on type overloading cannot propose these different constructors because the width and the height of a window are of the same type.

Languages without overloading nor n-ary functions traditionally use lists to collect optional and keyworded arguments. In addition to the runtime cost imposed by the list constructions, the called function has to dispatch, at runtime, over the list to set the parameters values. Furthermore, such a call cannot be statically typed anymore.

3.4 Containers and widget placement

Biglook uses special sort of widgets to enable user customized widget placements: the container class. A container is a widget that can embed other widgets. Those widgets are called the *children* of the container. For instance, a window such as the one defined line 4 of Figure 3, is a container, it “contains” the three other widgets. With the exception of windows, all widgets must be associated with a container in order to be visible on the screen. To associate a widget with a container, one have to set its parent slot (see lines 8, 12 & 18 of Figure 3). Biglook proposes several kind of containers: aligned containers (such as boxes and windows), grid containers, note pad containers, paned containers, containers with scrollbars, etc.

Let us present here two examples of containers. First, let us consider that we want to modify the interface of Figure 2. We want the buttons to be displayed horizontally instead of vertically

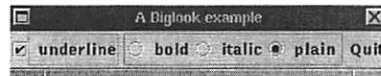


Figure 4: An horizontal layout

(see Figure 4). For that, we add a new container in the window, an horizontal box:

```
1:(define awin
2:  (instantiate::window
3:    (title "A Biglook example")))
4:
5:(define abox
6:  (instantiate::box
7:    (parent awin)
8:    (orientation 'horizontal)))
9:
10:(define acheck
11:  (instantiate::check-button
12:    (parent abox)
13:    (text "underline")))
14:...
```

For the second example, we combine containers to design complex interfaces. For instance, the interface of Figure 5 can be implemented as:

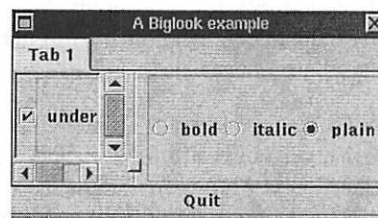


Figure 5: Several containers

```
1:(define awin
2:  (instantiate::window
3:    (title "A Biglook example")))
4:
5:(define atab
6:  (instantiate::notepad
7:    (parent awin)))
8:
9:(define apane
10:  (instantiate::paned
11:    (orientation 'horizontal)
12:    (parent atab)))
13:
14:(define ascroll
15:  (instantiate::scroll
16:    (parent apane)))
```

```

17:(define acheck
18:  (instantiate::check-button
19:    (parent ascroll)
20:    (text "underline")))
21:
22:(define aradio
23:  (instantiate::radio
24:    (parent apane)
25:    (border-width 2)
26:    (orientation 'horizontal)
27:    (texts '("bold" "italic" "plain"))))
28:
29:(define abutton
30:  (instantiate::button
31:    (parent awin)
32:    (relief 'flat)
33:    (text "Quit")))

```

Note that even if the interfaces of Figures 2 and 5 seem quite different, we only need to modify the *parent* slot of the *acheck* and *aradio* widgets to embed them in the new containers *atab* (line 6), *apane* (line 10) and *ascroll* (line 15).

3.5 Event Management

Biglook widgets allow the creation of complex GUIs with minimal efforts. In general, when building such interfaces, one of the main difficulties lies in trying to separate the code of the interface from the rest of the program. Making the GUI code independent from the rest of the application is important because:

- GUIs are often built on a trial-fail basis. It is hard to conceive an interface *ex-nihilo*, and it is generally after using it for a while that the elements of the GUI find their place. Keeping the code independent from the rest of the application allows the development of prototypes of the interface without nasty consequences on the other parts of the program.
- A given program can have several interfaces according to the device on which it is run (e.g. graphical screen, PDA, alphanumeric terminal). With an independent interface code, different interfaces can be connected to the same program.
- The GUI of an application can be constructed interactively by an interface builder. In such a case, it is preferable to keep the mechanically generated code separate from hand written parts of the application.

Graphical events (mouse click, key pressed, window destruction ...) can be associated with widgets by the means of the widgets event slot. This slot must contain an instance of the class *event-handler* which is defined as:

```

(class event-handler
  (configure::procedure (default ...))
  ;; window events
  (destroy::procedure (default ...))
  ...
  ;; mouse events
  (press::procedure (default ...))
  (enter::procedure (default ...))
  ...
  ;; keyboard events
  (key::procedure (default ...))
  ...))

```

Each slot of an event-handler is a procedure called a *call-back* that accepts one argument. When a graphical action occurs on a widget, the associated call-back is invoked passing it an *event descriptor*. Those descriptors are allocated by the Biglook runtime system. They are instances of the event class which is defined as:

```

(class event
  ;; the widget which receives the event
  (widget::widget read-only)
  ;; the button number or -1
  (button::int read-only)
  ;; the modifiers list (e.g. shift)
  (modifiers::pair-nil read-only)
  ;; the x position of the mouse
  (x::int read-only)
  ;; the y position of the mouse
  (y::int read-only)
  ;; the character pressed or -1
  (char::char read-only)
  ...))

```

So, modifying the example of Figure 3 for the Quit button to be aware of mouse button 1 clicks, we could write the code as follows:

```

1:(let* ((p (lambda (e)
2:  (if (= (event-button e) 1)
3:    (exit))))
4:  (evt (instantiate::event-handler
5:    (press p))))
6:  (with-access::button abutton (event)
7:    (set! event evt)))

```

That is, on line 4 we allocate *evt*, an instance of the *eventhandler* class. That event handler is connected to the button line 7. The event handler only reacts to mouse press events. When such an event is raised, the call-back line 1 is invoked,

its formal parameter *e* being bound to an instance of the event class. This function checks the button number of the raised event (line 2). When the first button is pressed the Biglook application exits.

It is possible to modify already connected call-backs. For instance, if we want the Quit button to emit a sound when it is pressed, we can write:

```
(with-access::button abutton (event)
  (with-access::event-handler event (press)
    (let ((olde press))
      (set! press
        (lambda (e)
          (beep) (olde e)))))))
```

Since widgets call-backs are plain Scheme closures, they can be manipulated as first class objects, as in this example where new call-backs capture the values of the old call-backs in order to reuse them.

3.6 Reflection on event handling

Most modern widget toolkits (with the exception of Qt [5]) use a call-back framework. That is, user commands are associated with specific events (such as mouse click, mouse motion, keyboard inputs, ...). When an event is raised, the user command is invoked. We think that the closure mechanism is the most simple and efficient way to implement call-backs even if some alternatives exist.

The rest of this section discusses how call-backs can be implemented depending on the features provided by the host language used to implement a graphical toolkit.

3.6.1 Languages that support functions without environment

ISO-C [16] supports global functions but no local functions. A C function is always top-level and may only access its parameters and the set of global variables. C functions have no definition environment. However, without an environment, a call-back is extremely restricted. In particular, a call-back is likely to access the widget that owns it. In GTK+ (a C toolkit) when a call-back is as-

sociated with an event, an optional value may be specified that will be passed to the call-back when the event is raised. This user value actually *is* the environment of the call-back. GTK+ mimics closures with its explicit parameter-passing scheme. We may notice that the allocation and the management of the closure environment is in charge of the client application.

3.6.2 Languages with classes

For languages with classes such as Java, another strategy can be used. Call-backs may be implemented using class member functions. Member functions may access the object and the object's attributes for which they are invoked. Member functions look like closures. However, member functions are not closures because they are associated with classes. In other words, all the instances of a class share the implementation of all their member functions. That is, different call-back implementations require different class declarations. For instance, if one wants to implement a button with a call-back printing a plain message and another one emitting a sound, two classes have to be defined. These class declarations turn out to be an hindrance to simplicity and readability. In addition, if several events must be handled by one widget, this technique turns out to be impractical because it is not possible to define a new class for each kind of events the widget must react to (mouse-1, mouse-2, mouse-3, shift-mouse-1, ctrl-mouse-1, shift-ctrl-mouse-1, ...).

To avoid these extra class definitions, Java has introduced inner classes. An inner class is a class defined inside another class; it may be anonymous. Because in GUI programming inner classes are used to implement call-backs and as they are numerous, Java proposes a new syntax that enables within a single expression, to declare and to instantiate an inner classe. For instance:

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    a user code that may reference
    current lexical bindings
  }
})
```

The expression `new ActionListener...` deserves some explanation: 1) it declares an anonymous inner class that 2) implements the in-

terface ActionListener and 3) it creates one unique instance of that new class that is sent to the method addActionListener of the Button instance. That is, anonymous inner classes are the exact Java implementation of closures. It is worth pointing out that Scheme syntax for closures is far more compact than the Java one.

3.6.3 Closures

Closures are central to GUI programming because they are one of the most natural way to implement call-backs. As we have seen, all call-back based toolkits offer a mechanism similar to closures. It can be member functions or anonymous Java classes or extra parameter passed to C functions. However, we have found that solutions of non-functional programming languages are not as convenient as Scheme's lambda expression because either the user is responsible of the construction of the object representing the closure or extra syntax is introduced.

```
1:(define tree1
2:  (instantiate::tree
3:    (parent apane)
4:    (root object)
5:    (node-label class-name)
6:    (node-children class-subclasses)))
7:(define tree2
8:  (instantiate::tree
9:    (parent apane)
10:   (root "/")
11:   (node-label (lambda (x) x))
12:   (node-children directory->list)))
```

Figure 6: Two Biglook trees

Not only call-backs are easily implemented by the means of closures but we have also found that closures are handy to implement pre-existing data structure visualization. Consider the screen shot, Figure 7. It is made of two Biglook trees. A Biglook tree is a visualization of an existing data structure. That is, a Biglook tree does not contain data by itself. It only *visualizes* an existing structure. A Biglook tree is defined by three slots: *i*) the root of the tree (*root*), *ii*) a function that extracts a string which stands for the label of a node (*node-label*), *iii*) a function that computes the children list of a node (*node-children*). The declaration of the trees of Figure 7 are given Fig-

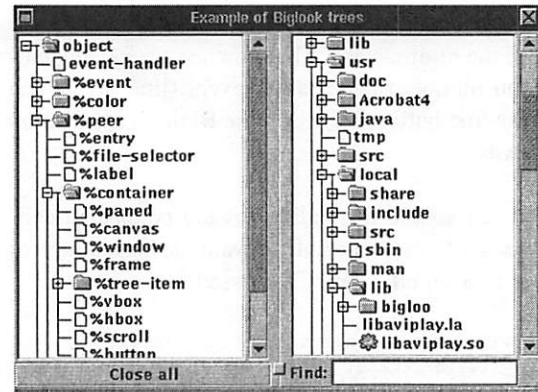


Figure 7: Two Biglook trees

ure 6, lines 2 and 8. The first tree (*tree1*) is a class tree, its root is the Scheme object class denoting the root of the inheritance tree (line 4). The second tree (*tree2*) is a file tree. Its root is *"/"*, the root of the file system (line 10). To compute the name of a node representing a class it is only required to extract the name of that class (line 5). The name of a node representing a directory is the node itself since the node is a string (line 11). The children list of a class is computed using the Bigloo library function *class-subclasses* (line 6). The children list of a directory is computed by the library function *directory->list* (line 12). As one may notice "hooking" a tree to a data structure is a simple task. The resulting program is compact. We think that this compactness is another strength of Scheme closures.

In general, the Biglook API enables small source codes for GUIs. On a typical graphical interface we have found that the Biglook source is about twice smaller than the same interface implemented in Tcl/Tk and about four times smaller than the interface implemented in C with GTK+.

4 Implementing Biglook

In this section, we present the overall Biglook architecture and implementation. Then, we detail the role of *virtual slots*.

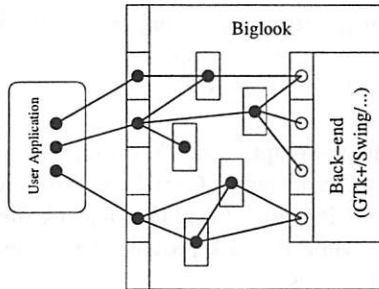


Figure 8: The Biglook software architecture

4.1 Library Architecture

The Biglook library is implemented on top of a native *back-end* toolkit (currently a GTK+ back-end and a Swing back-end are available). It takes advantage of the efficiency of the *back-end* low level operations and it imposes a low memory overhead (about 15% of additional allocations). The difference in execution time is marginal. As a consequence, Biglook can be used to implement applications using complex graphical interfaces such as the one presented Figure 1. Interested readers can find precise performance evaluation of these implementations in a previous technical report [12].

Because Biglook makes few assumptions about the underlying toolkit, re-targeting its implementation to another back-end is generally possible. To be a potential Biglook back-end, a toolkit must provide: *i*) a way to identify a particular component of the interface. Of course all the toolkits provide this, even if the representation used can differ, such as an integer, a string, a function, and so on. *ii*) an event manager that does not hide any events. All the toolkits we have tried satisfy this criteria (Tk, GTK+ and Swing). In addition, if a back-end lacks some widgets, Biglook implements them using primitive widgets. Figure 8 shows the underlying architecture of the Biglook toolkit.

Actual graphical toolkits generally support these prerequisites and, as such, can be used as potential Biglook back-end. We will see in Section 4.2 that using virtual slots allows the building of the rest of the library on this minimal basis.

4.2 Virtual Slots and Biglook

A simple widget is a widget that is directly mapped into a builtin widget. All simple widgets are implemented according to the same framework: they inherit from the widget class and they define user customization options. These options are implemented using virtual slots. We present here a possible implementation of the `label` class. For the sake of simplicity, we assume that this class extends the widget class with only one additional slot: the text slot that specifies the label text.

```
(class widget::object
  (builtin-widget read-only)
  ...)

(class label::widget
  (text
    (get
      (lambda (o::label)
        (with-access::label o (builtin)
          (<builtin-label-text> builtin))))))
  (set
    (lambda (o::label v::procedure)
      (with-access::label o (builtin)
        (<builtin-label-text-set!> builtin
          v))))))
```

Implementing the text slot requires virtual slots. Its getter and setter functions directly interact with the back-end toolkit. Virtual slots are used to establish the connection between Biglook user point of view of widgets and their native implementation. Virtual slots are used to provide an object oriented class-based API to Biglook, independent of the back-end.

Now that the slots of a label widget are defined, we must define how such a widget has to be initialized. The class initialization specific code is given to the system via the generic function `realize-widget`. For each class of the library a method overrides this generic function and must call the back-end to create the graphical object associated with the class. For a label, the method we need to write is:

```
(define-method (realize-widget o::label)
  (with-access::label o (builtin parent)
    (set! builtin
      (<builtin-make-label> parent))))
```

This method creates a builtin label via the low level `<builtin-make-label>` function and stores

the result in the builtin slot. This slot creates the link between the Biglook toolkit and the back-end.

5 Related work

Many functional languages are connected to widget libraries especially to the Tk toolkit. Few of them use object-oriented programming except in the Scheme world, we can cite mainly STk, SWL and MrEd.

5.1 Scheme widget libraries

STk [11] is a Scheme interpreter extended with the Tk graphical toolkit. To some extent, STk is the ancestor of Biglook, since it was developed by one of the authors of this paper. However, STk is tightly coupled to the Tk toolkit and even if this toolkit is presented to the user through an object oriented API as in Biglook, no provision was made to be independent from this back-end.

SWL is a contribution to the Petite Chez Scheme system [6]. It relies on an interpreter and uses Tk as back-end. In SWL, native Tk widgets are mapped to Chez Scheme classes and in this respect this toolkit is similar to the Biglook or STk libraries. However, SWL implementation is very different from the one used by those libraries since SWL widgets explicitly *talk* with a regular Tcl/Tk evaluator.

MrEd [9], a part of the DrScheme project [7], is a programming environment that contains an interpreter, a compiler and other various programming tools such as browser, debugger, etc. The back-end toolkit used by MrEd is wxWindow [27], a toolkit that is available under various platforms (Unix, Windows, etc.). As STk, this toolkit is completely dependent of its back-end.

5.2 Other functional languages widget libraries

Other functional languages provide graphical primitives using regular functions. The main con-

tribution for strict functional programming languages has been developed for the Caml programming language [30].

The first attempt, CamlTk, is quickly surveyed in [23]. The design of CamlTk is different from the one of Biglook. CamlTk *binds* Tk functions in Caml while Biglook provides an original API made of classes.

Recently a new widget library based on GTK+ has been proposed for Caml. No article describes that connection. However, some work has been described to add keyword parameters to Caml in order to help the connection with widget libraries [10]. The philosophy of that work differs from ours because that new library makes the GTK+ API available from Caml. No attempts are made to present a neutral API as we did for Biglook.

Programming graphical user interfaces with lazy languages is far more challenging than with strict functional languages. The problem is to tame the imperative aspects of graphical I/O in such languages [18, 28]. Several solutions have been proposed: Fudget by Carlsson and Hallgren [2, 3], Haggis by Finne and Peyton Jones [8], TkGofer by Vullings and Claessen [4] and, more recently the extension of the former TclHaskell library: FranTk by Sage [24].

Conclusion

In this paper we have presented Biglook, a widget library for the Bigloo system. The architecture of Biglook enables different ports. Currently two ports are available: a GTK+ port and a Swing port. Biglook source code can be indifferently linked against the two libraries. Biglook API uses an object oriented programming style to handle graphical objects and a functional oriented programming style to implement the interface reactivity. We have found that combining the two programming styles enables more compact implementations for GUIs than most of the other graphical toolkits.

Acknowledgments

Many thanks to Keith Packard, Jacques Garrigue, Didier Remy, Peter Sander, Matthias Felleisen, Simon Peyton Jones and to Céline for their helpful feedbacks on this work.

References

- [1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. **Common lisp object system specification**. In *special issue*, number 23 in SIGPLAN Notices, September 1988.
- [2] M. Carlsson and T. Hallgren. **FUDGETS - A Graphical User Interface in a Lazy Functional Language**. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.
- [3] M. Carlsson and T. Hallgren. **Fudgets — Purely Functional Processes with applications to Graphical User Interfaces**. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden, March 1998.
- [4] K. Claessen, T. Vullingsh, and E. Meijer. **Structuring Graphical Paradigm in TkGofer**. In *Int'l Conf. on Functional Programming*, 1997.
- [5] M. Dalheimer. **Programming with Qt**. O'Reilly, 1st edition, april 1999.
- [6] K. Dybvig. **Chez Scheme User's Guide**. Cadence Research Systems, 1998.
- [7] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. **The DrScheme Project: An Overview**. *SIGPLAN Notices*, 1998.
- [8] S. Finne and S. Peyton Jones. **Composing Haggis**. In *Fifth Eurographics Workshop on Programming Paradigm for Computer Graphics*, Maastricht, NL, September 1995.
- [9] M. Flatt, R. Findler, S. Krishnamuryhi, and M. Felleisen. **Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine)**. In *Int'l Conf. on Functional Programming*, Paris, France, 1999.
- [10] J. Furuse and J. Garrigue. **A label-selective lambda-calculus with optional arguments and its compilation method**. Technical Report RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.
- [11] E. Gallesio. **STk Reference Manual**. Technical Report RT 95-31a, I3S-CNRS/Univ. of Nice–Sophia Antipolis, July 1995.
- [12] E. Gallesio and M. Serrano. **Graphical user interfaces with Biglook**. Technical Report I3S/RR-2001-13-FR, I3S-CNRS/Univ. of Nice–Sophia Antipolis, September 2001.
- [13] A. Goldberg and D. Robson. **Smalltalk-80: The Language and Its Implementation**. Addison-Wesley, 1983.
- [14] J. Gosling, B. Joy, and G. Steele. **The Java™ Language Specification**. Addison-Wesley, 1996.
- [15] J. Gosling, F. Yellin, and the Java Team. **The Java™ Application Programming Interface, Volume 2: Window Toolkit and Applets**. Addison-Wesley, 1996.
- [16] ISO/IEC. **9899 Programming Language - C**. Technical Report DIS 9899, ISO, July 1990.
- [17] R. Kelsey, W. Clinger, and J. Rees. **The Revised(5) Report on the Algorithmic Language Scheme**. *Higher-Order and Symbolic Computation*, 11(1), September 1998.
- [18] R. Noble and C. Runciman. **Functional Languages and Graphical User Interfaces – a review and a case study**. Technical Report 94-223, Department of computer Science, University of York, February 1994.
- [19] J. Ousterhout. **Tcl and the Tk toolkit**. Addison-Wesley, 1994.
- [20] H. Pennington. **Gtk+/Gnome Application Development**. New Riders Publishing, 1999.
- [21] C. Queinnec. **Designing MEROON V3**. In *Workshop on Object-Oriented Programming in Lisp*, 1993.
- [22] D. Rémy and J. Vouillon. **Objective ML: A simple object-oriented extension of ML**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.
- [23] F. Rouaix. **A Web navigator with applets in Caml**. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28:7–11, pages 1365–1371. Elsevier, May 1996.
- [24] M. Sage. **FranTk – A declarative GUI language for Haskell**. In *Int'l Conf. on Functional Programming*, Montral, Qubec, Canada, September 2000.
- [25] M. Serrano. **Bigloo user's manual**. RT 0169, INRIA-Rocquencourt, France, December 1994.
- [26] M. Serrano. **Wide classes**. In *Proceedings ECOOP'99*, pages 391–415, Lisbon, Portugal, June 1999.
- [27] J. Smart. **wxWindows toolkit Reference Manual**. available at <http://web.ukonline.co.uk/julian-smart/wxwin>, 1992.
- [28] T. Vullings, D. Tuijnman, and V. Schulte. **Lightweight GUIs for Functional Programming**. In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, 1995.
- [29] K. Walrath and M. Campione. **The JFC Swing Tutorial: A Guide to Constructing GUIs**. Addison-Wesley, July 1999.
- [30] P. Weis and *al.* **The CAML Reference manual**. Technical Report 121, INRIA-Rocquencourt, 1991.

An Implementation of Scheduler Activations on the NetBSD Operating System

Nathan J. Williams
Wasabi Systems, Inc.
nathanw@wasabisystems.com

Abstract

This paper presents the design and implementation of a two-level thread scheduling system on NetBSD. This system provides a foundation for efficient and flexible threads on both uniprocessor and multiprocessor machines. The work is based on the scheduler activations kernel interface proposed by Anderson et al. [1] for user-level control of parallelism in the presence of multiprogramming and multiprocessing.

First, as motivation, Section 2 describes traditional thread implementations. Scheduler activations are explained in the context of two-level thread implementations, and then described in detail. Section 4 gives the interface that the scheduler activations system presents to programs, and Section 5 follows with details about the kernel implementation behind the interface. The thread library built on this interface is described in Section 6, and the performance of the system and the thread library is examined and compared to other libraries in Section 7. Finally, Section 8 concludes and considers directions for future work.

1 Introduction

Thread programming has become a popular and important part of application development. Some programs want to improve performance by exploiting concurrency; others find threads a natural way to decompose their application structure. However, the two major types of thread implementation available – user threads and kernel threads – both have significant drawbacks in overhead and concurrency that can limit the performance of applications.

A potential solution to this problem exists in the form of a hybrid, two-level thread system known as *scheduler activations* that divides the work between the kernel and the user levels. Such a system has the potential to achieve the high performance of user threads while retaining the concurrency of kernel threads.

The purpose of this paper is to describe the design and implementation of such a two-level system and associated thread library in NetBSD, show that the speed of thread operations is competitive with user thread implementations, and demonstrate that it can be implemented without hurting the performance of unrelated parts of the system.

2 Thread Systems

Historically, there have been two major types of thread implementations on Unix-like systems, with the essential differences being the participation of the kernel in the thread management. Each type has significant drawbacks, and much work has gone into finding a compromise or third way.

The first type of thread system is implemented purely at user-level. In this system, all thread operations manipulate state that is private to the process, and the kernel is unaware of the presence of the threads. This type of thread system is often known as “N:1” thread system because the thread implementation maps all of the N application threads onto a single kernel resource.

Examples of this type of thread system include the GNU PTH thread library [6], FSU Pthreads, PTL2 [7], Provenzano’s “MIT pthreads” library [8], and the original DCE threads package that formed so much of the basis for the POSIX thread effort [4]. Some large software packages contain their own thread system, especially those that were originally written to support platforms without native thread support (such as the “green threads” in Sun’s original Java implementation). All of the *BSDs

currently have one of these user-level packages as their primary thread system.

User-level threads can be implemented without kernel support, which is useful for platforms without native thread support, or applications where only a particular subset of thread operations is needed. The GNU PTH library, for example, does not support preemptive time-slicing among threads, and is simpler because of it. Thread creation, synchronization, and context switching can all be implemented with a cost comparable to an ordinary function call.

However, operations that conceptually block a single thread (blocking system calls such as `read()`, or page faults) instead block the entire process, since the kernel is oblivious to the presence of threads. This makes it difficult to use user-level threads to exploit concurrency or provide good interactive response. Many user-level thread packages partially work around this problem by intercepting system calls made by the application and replacing them with non-blocking variants and a call to the thread scheduler. These workarounds are not entirely effective and add complexity to the system.

Additionally, a purely user-level thread package can not make use of multiple CPUs in a system. The kernel is only aware of one entity that can be scheduled to run – the process – and hence only allocates a single processor. As a result, user-level thread packages are unsuitable for applications that are natural fits for shared-memory multiprocessors, such as large numerical simulations.

At the other end of the thread implementation spectrum, the operating system kernel is aware of the threaded nature of the application and the existence of each application thread. This model is known as the “1:1” model, since there is a direct correspondence between user threads and kernel resources. The kernel is responsible for most thread management tasks: creation, scheduling, synchronization, and disposal. These kernel entities share many of the resources traditionally associated with a process, such as address space and file descriptors, but each have their own running state or saved context.

This approach provides the kernel with awareness of the concurrency that exists within an application. Several benefits are realized over the user-thread model: one thread blocking does not impede the progress of another, and multiprocessor parallelism can be exploited. But there are problems here as well. One is that the overhead of thread operations is high: since they are managed by the kernel, operations must be performed by re-

questing services of the kernel (usually via system call), which is a relatively slow operation.¹ Also, each thread consumes kernel memory, which is usually more scarce than user process memory. Thus, while kernel threads provide better concurrency than user threads, they are more expensive in time and space. They are relatively easy to implement, given operating system support for kernel execution entities that share resources (such as the `clone()` system call under Linux, the `spawn()` system call under IRIX, or the `_lwp_create()` system call in Solaris). Many operating systems, including Linux, IRIX, and Windows NT, use this model of thread system.

Since there are advantages and disadvantages of both the N:1 and 1:1 thread implementation models, it is natural to attempt to combine them to achieve a balance of the costs and benefits of each. These hybrids are collectively known as “N:M” systems, since they map some number N of application threads onto a (usually smaller) number M of kernel entities. They are also known as “two-level” thread systems, since there are two parties, the kernel and the user parts of the thread system, involved in thread operations and scheduling. There are quite a variety of different implementations of N:M thread systems, with different performance characteristics. N:M thread systems are more complicated than either of the other models, and can be more difficult to develop, debug, and use effectively. Both AIX and Solaris use N:M thread systems by default.²

In a N:M thread system, a key question is how to manage the mapping of user threads to kernel entities. One possibility is to associate groups of threads with single kernel entities; this permits concurrency across groups but not within groups, reaching a balance between the concurrency of N:1 and 1:1 systems.

The *scheduler activations* model put forward by Anderson et al. is a way of managing the N:M mapping while maintaining as much concurrency as a 1:1 thread system. In this model, the kernel provides the application with the abstraction of virtual processors: a guarantee to run a certain number of application threads simultaneously on CPUs. Kernel events that would affect the number of running threads are communicated directly to the application in a way that maintains the number of virtual processors. The message to the application informs it

¹The DG/UX operating system prototyped an implementation that took advantage of the software state saving of their RISC processor to permit fast access to simple kernel operations. This technique has not been widely adopted.

²Sun Solaris now also ships with a 1:1 thread library; application developers are encouraged to evaluate both thread libraries for use by their application.

of the state that has changed and the context of the user threads that were involved, and lets the user-level scheduler decide how to proceed with the resources available.

This system has several advantages: like other M:N systems, kernel resource usages is kept small in comparison to the number of user-level threads; voluntary thread switching is cheap, similar to user-level threads, and like 1:1 systems, an application's concurrency is fully maintained. Scheduler activations have been implemented for research purposes in Taos [1], Mach 3.0 [2], and BSD/OS [9], and adopted commercially in Digital Unix [5] (now Compaq Tru64 Unix).

The scheduler activations system shares with other M:N systems all of the problems of increased complexity over 1:1 systems. Additionally, there is concern that the problems addressed by scheduler activations are not important problems in the space of threaded applications. For example, making thread context switches cheap is of little value if thread-to-thread switching is infrequent, or if thread switching occurs as a side effect of heavyweight I/O operations.

Implementing scheduler activations for NetBSD is attractive for two major reasons. First, NetBSD needs a native thread system which is preemptive and has the ability to exploit multiprocessor computer systems. Second, this work makes a scheduler activations interface and implementation available in an open-source operating system for continued research into the utility and viability of this intuitively appealing model.

3 Scheduler Activations

As described by Anderson et al., the scheduler activations kernel provides the application with a set of virtual processors, and then the application has complete control over what threads to run on each of the virtual processors. The number of virtual processors in the set is controlled by the kernel, in response to the competing demands of different processes in the system. For example, an application may express to the kernel that it has enough work to keep four processors busy, while a single-threaded application is also trying to run; the kernel could allocate three processors to the set of virtual processors for the first application, and give the fourth processor to the single-threaded program.

In order for the application to be able to consistently use these virtual processors, it must know when threads have

blocked, stopped, or restarted. For user-level operations that cause threads to block, such as sleeping for a mutex or waiting on a condition variable, the thread that is blocking hands control to the thread library, which can schedule another thread to run in the usual manner. However, kernel-level events can also block threads: the `read()` and `select()` system calls, for example, or taking a fault on a page of memory that is on disk. When such an event occurs, the number of processors executing application code decreases. The scheduler activations kernel needs to tell the application what has happened and give it another virtual processor. The mechanism that does this is known as an *upcall*.

To perform an upcall, the kernel allocates a new virtual processor for the application and begins running a piece of application code in this new execution context. The application code is known as the *upcall handler*, and it is invoked similarly to a traditional signal handler. The upcall handler is passed information that identifies the virtual processor that stopped running and the reason that it stopped. The upcall handler can then perform any user-level thread bookkeeping and then switch to a runnable thread from the application's thread queue.

Eventually the thread that blocked in the kernel will unblock and be ready to return to the application. If the thread were to directly return, it would violate two constraints of scheduler activations: the number of virtual processors would increase, and the application would be unaware that the state of the thread had changed. Therefore, this event is also communicated with an upcall. In order to maintain the number of virtual processors, the thread currently executing on one of the application's processors is preempted. The upcall is then run in the context of that virtual processor, carrying notifications of both the first thread returning from the kernel and the second thread being preempted. The upcall handler, knowing all of this, can then decide which thread to run next, based on its own processing needs. Typically, this would choose the highest priority thread, which should be one of the two involved in the upcall notification.

There are other scheduling-related events that are communicated by upcall. A change in the size of the virtual processor set must be communicated to the application so that it can either reschedule the thread running on a removed processor, or schedule code to run on the new processor. Traditional POSIX signals, which would normally cause a control-flow change in the application, are communicated by upcall. Additionally, a mechanism is provided for an application to invoke an upcall on another processor, in order to bring that processor back under control of the thread engine (in case thread en-

gine code running on processor 1 decides that a different, higher-priority thread should start running on processor 2).

4 Kernel Interface

The application interface to the scheduler activations system consists of system calls. First, the `sa_register()` call tells the kernel what entry point to use for a scheduler activations upcall, much like registering a signal handler. Next, `sa_setconcurrency()` informs the kernel of the level of concurrency available in the application, and thus the maximum number of processors that may be profitably allocated to it. The `sa_enable()` call starts the system by invoking an upcall on the current processor. While the application is running, the `sa_yield()` and `sa_preempt()` calls allow an application to manage itself by giving up processors and interrupting other processors in the application with an upcall.

4.1 Upcalls

Upcalls are the interface used by the scheduler activations system in the kernel to inform an application of a scheduling-related event. An application that makes use of scheduler activations registers a procedure to handle the upcall, much like registering a signal handler. When an event occurs, the kernel will take a processor allocated to the application (possibly preempting another part of the application), switch to user level, and call the registered procedure.

The signature of an upcall is:

```
void sa_upcall(int type,
               struct sa_t *sas[],
               int events,
               int interrupted,
               void *arg);
```

The type argument indicates the event which triggered the upcall. The types and their meanings are described below.

The `sas` field points to an array of pointers to `struct sa_t` which describe the activations involved in this event. The first element of this array, `sas[0]`, points

to the `sa_t` of the running activation. The next elements of the array (`sas[1]` through `sas[events]`) describe the activations directly involved in an event. The value of `events` may be zero, in which case none of the array elements are used for this purpose. The remaining elements of the array (`sas[events+1]` through `sas[events+interrupted]`) describe the activations that were stopped in order to deliver this upcall; that is, the “innocent bystanders”. The value of `interrupted` may be zero, in which case none of the elements are used for this purpose.

Upcalls are expected to switch to executing application code; hence, they do not return.

The set of events that generates upcalls is as follows:

- **SA_UPCALL_NEWPROC** This upcall notifies the process of a new processor allocation. The first upcall to a program, triggered by `sa_enable()`, will be of this type.
- **SA_UPCALL_PREEMPTED** This upcall notifies the process of a reduction in its processor allocation. There may be multiple “event” activations if the allocation was reduced by several processors.
- **SA_UPCALL_BLOCKED** This upcall notifies the process that an activation has blocked in the kernel. The `sa_context` field of the event should not be continued until a **SA_UPCALL_UNBLOCKED** event has been delivered for the same activation.
- **SA_UPCALL_UNBLOCKED** This upcall notifies the process that an activation which previously blocked (and for which a **SA_UPCALL_BLOCKED** upcall was delivered) is now ready to be continued.
- **SA_UPCALL_SIGNAL** This upcall is used to deliver a POSIX-style signal to the process. If the signal is a synchronous trap, then `event` is 1, and `sas[1]` points to the activation which triggered the trap. For asynchronous signals, `event` is 0. The `arg` parameter points to a `siginfo_t` structure that describes the signal being delivered.
- **SA_UPCALL_USER** This upcall is delivered when requested by the process itself with `sa_preempt()`. The `sas[1]` activation will be the activation specified in the call.

If the last processor allocated to a process is preempted, then the application can not be informed of this immediately. When it is again allocated a processor, an upcall is

used to inform the application of the previous preemption and the new allocation, so that the user-level scheduler can reconsider the application's needs.

The low level upcall mechanism is similar to signal delivery. A piece of code, known as the upcall trampoline, is copied to user space at the start of program execution. To invoke an upcall in the application, the kernel copies out the upcall arguments into user memory and registers, switches to user level with the arguments to the upcall and the address of the upcall entry point available, and starts running at the trampoline code, which calls the upcall routine.

4.2 Stacks

Upcall code, like any application code written in C, needs a stack for storage of local variables, return addresses, and so on. Using the stack of a preempted thread is not always possible, as there is no preempted thread in the case of new processor allocations. Also, using the stack of a preempted thread makes thread management more difficult, because that thread can not be made runnable again until the upcall code is exiting, or another processor might start running it and overwrite the upcall handler's stack area. Therefore, upcalls must be allocated their own set of stacks. The `sa_stacks()` system call gives the kernel a set of addresses and sizes that can be used as stacks for upcalls. Since the kernel does not keep track of when an upcall handler has finished running, the application must keep track of which stacks have been used for upcalls, and periodically call `sa_stacks()` to recycle stacks that have been used and make them available again. By batching these stacks together, the cost of the `sa_stacks()` system call is amortized across a number of upcalls.

4.3 Signals

The POSIX thread specification has a relatively complicated signal model, with distinctions drawn between signals directed at a particular thread and signals directed at the process in general, per-thread signal blocking masks but process-level signal actions, and interfaces to wait for particular signals at both the process and thread level. The method of handling signals under scheduler activations must permit a thread package to implement the POSIX signal model.

Since the kernel does not know about specific threads, it

can not maintain per-thread signal masks and affect per-thread signal delivery. Instead, signals are handed to the application via the upcall mechanism, with the `arg` parameter pointing to a `struct siginfo_t`. The user thread library can use this to invoke the signal handler in an appropriate thread context. In order to do this, though, the user thread code must intercept the application's calls to `sigaction()` and maintain the table of signal handlers itself.

5 Kernel Implementation

This section describes the changes needed to implement scheduler activations in the NetBSD kernel, including the separation of traditional process context from execution context, the mechanics of adapting the kernel execution mechanics to maintaining the invariants of scheduler activations, and the separation of machine-dependent and machine-independent code in the implementation.

5.1 LWPs

Most of the systems where scheduler activations has been implemented to date (Taos, Mach, and the Mach-inspired Digital Unix) have kernels where a user process is built out of a set of kernel entities that each represent an execution context. This fits well with scheduler activations, where a single process can have several running and blocked execution contexts. Unfortunately, the NetBSD kernel, like the rest of the BSD family, has a monolithic process structure that includes execution context.

The implementation of scheduler activations on BSD/OS by Seltzer and Small [9] dealt with this problem by using entire process structures for execution context. This had substantial problems. First, the amount of kernel memory that is used for each activation is larger than necessary. Second, using multiple processes for a single application causes a great deal of semantic difficulty for traditional process-based interfaces to the kernel. Applications like `ps` and `top` will show multiple processes, each apparently taking up the same amount of memory, which often confuses users attempting to understand the resource usage of their system. Sending POSIX signals is an action defined on process IDs, but targeting a process which is a sub-part of an application conflicts with the POSIX threads

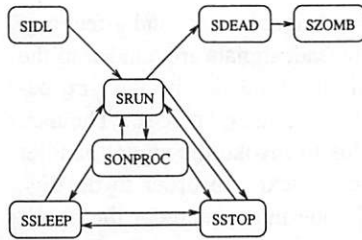


Figure 1: Original NetBSD Process States

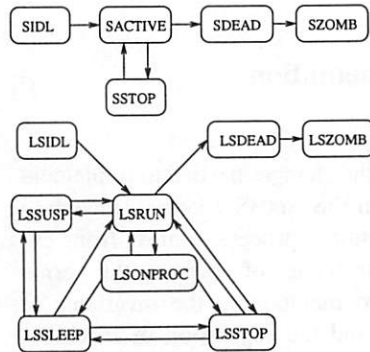


Figure 2: New NetBSD Process and LWP States

specification that an entire application should respond to signals, and that any thread may handle an incoming signal.³ Finally, complexity must be introduced in the kernel to synchronize per-process data structures such as file descriptor lists, resource limits, and credentials.

Therefore, the first stage in implementing scheduler activations was separating process context from execution context in NetBSD. This was a slow but largely mechanical undertaking. The parts of the classic BSD `struct proc` that related to execution context were relocated to a new structure, `struct lwp` (LWP for “light-weight process”, following Solaris and others). This included scheduling counters and priority, the run state and sleep fields, the kernel stack, and space for execution-context-specific machine-dependent fields. The process state values were reduced to those that represent the state of an entire process, and the execution-related process state values were changed to LWP state values.

Following this was an audit of every use of a variable of type `struct proc` within the kernel to determine whether it was being used for its process context or execution context. Execution context turned out to be the

³The LinuxThreads pthread implementation, while not based on scheduler activations, uses entire processes as threads and has all of these problems. “Why doesn’t kill work on my threaded application?” is a frequently heard question in Linux and thread programming forums.

prevalent use of such variables, especially the global variable `curproc`. The conversion process consisted of replacing variables like “`struct proc *p`” with “`struct lwp *l`”, and changing any code that actually referred to the process-level state to access it indirectly via a pointer in `struct lwp`. The scheduler was converted to handle scheduling LWPs rather than processes; the `fork()` system call was changed to create new LWPs for new processes, and the kernel “reaper” subsystem was adjusted to remove dead LWPs as well as dead processes.

Once this conversion was complete, the next stage was to permit the existence and concurrent execution of several LWPs within a single process. While the scheduler activations system will not have LWPs in a single process time sliced against each other, doing so was a good way of testing the LWP infrastructure, and may also be useful for binary compatibility with systems that use multiple LWPs per process. For the interface, several Solaris LWP functions were adopted: `_lwp_create()`, `_lwp_self()`, `_lwp_exit()`, `_lwp_suspend()`, and `_lwp_continue()`.

Two areas of the kernel were significantly affected by this: signal delivery and process exit. Previously, signal delivery was a straightforward switch on the state of the process. Now, with multiple LWPs and a large combination of possible states, the signal delivery code must iterate over the LWPs in order to find one that can accept the signal. Signals with actions that affect the state of all LWPs, such as `SIGSTOP` and `SIGCONT`, must also iterate over the LWPs and stop or continue each one as appropriate.

Process exit is complicated by the need to clean up all LWPs, not just the one that invokes `exit()`. The first LWP to attempt to exit must wait for all other LWPs to clean themselves up before cleaning up the process context. The kernel `_lwp_wait()` primitive requires some help to do this, especially in the presence of other LWPs that may be in similar sleep loops in the kernel. They must be coerced to quit their sleep loops while permitting the exiting lwp to continue in its loop. This is done by making `tsleep()` exit when a process is trying to exit (as noted by the `P_WEXIT` flag in `struct proc`), but providing a flag `P_NOEXITERR` that makes the exiting LWP’s wait loop ignore `P_WEXIT`.

The machine-dependent parts of the NetBSD kernel each require some porting work to make them work with LWPs. Some of this is straightforward: implementing the machine-dependent back ends for the `getcontext()` and `setcontext` system

calls and changing some flags from `P_FLAGNAME` to `L_FLAGNAME`. More involved is splitting the machine-dependent parts of the old `struct proc` into the new `struct proc` and `struct lwp`. For example, on the i386, the TSS selector needs to be LWP-specific, but the pointer to the system call entry point needs to be proc-specific. On some architectures, such as the PowerPC, the machine-dependent part of `struct proc` becomes empty.

Finally, some delicate work is required in the code that implements the process context switch, which is usually written in assembler. The existing `cpu_switch()` function, which picks another process to run from the run queue, must be modified to return a flag indicating whether or not it switched to another process. This is used by the scheduler activations code to determine if a preemption upcall needs to be sent. A variant of this routine called `cpu_preempt()` must be implemented, which takes a new LWP to switch to, instead of picking one from the run queue. This is used by the scheduler activations code to continue executing within the same process when one LWP is blocked.

5.2 Scheduler Activations

The kernel implementation of the actual scheduler activations system is centered on a routine, `sa_upcall()`, which registers the need for an upcall to be delivered to a process, but does not actually modify the user state. This routine is used, for example, by system calls that directly generate upcalls, such as `sa_preempt()` and `sa_enable()`.

The most interesting work occurs when a process running with scheduler activations enabled is in the kernel and calls the `tsleep()` function, which is intended to block the execution context and let the operating system select another process to run. Under the scheduler activations philosophy, this is the moment to send an upcall to the process on a new virtual processor so that it can continue running; this is handled by having `tsleep()` call a function called `sa_switch()` instead of the conventional `mi_switch()`. The mechanics of this are complicated by resource allocation issues; allocating a new activation LWP could block on a memory shortage, and since blocking means calling `tsleep()` again, recursion inside `tsleep` would result. To avoid this problem, a spare LWP is pre-allocated and cached when scheduler activations are enabled, and each such LWP allocates another as its first action when it runs, thus ensuring that no double-sleep recursion occurs.

The `sa_switch()` code sets the LWP to return with a “blocked” upcall, and switches to it. The new LWP exits the kernel and starts execution in the application’s registered upcall handler.

When the LWP that had called `tsleep()` is woken up, it wakes up in the middle of `sa_switch()`. The switch code sets the current LWP to return with an “unblocked” upcall, potentially including the previously running LWP as an “interrupted” activation if it belongs to the same process.

The other important upcall is the “preempted” upcall. The scheduling code in NetBSD periodically calls a routine to select a new process, if another one exists of the same or higher priority. That routine calls `sa_upcall()` if it preempts a scheduler activations LWP.

The upcall delivery is done just before crossing the protection boundary back into user space. The arguments are copied out by machine-dependent code, and the process’s trap frame is adjusted to cause it to run the upcall-handling code rather than what was previously active.

5.3 Machine dependence

The architecture-dependent code needed to support scheduler activations in NetBSD amounts to about 4000 lines of changed code per architecture. The bulk of that is the mechanical replacement of `struct proc` references with `struct lwp` references. About 500 lines of new code is necessary to implement `getcontext()` and `setcontext`, the machine-dependent upcall code, and the `cpu_preempt()` function. To date, the work to make an architecture support scheduler activations has been done by the author on two architectures, and by other people on five others. None of the volunteers who did this porting work reported any significant problems in the interface between machine-dependent and machine-independent scheduler activations code.

6 Thread Implementation

The principal motivation for the scheduler activations system is to support user-level concurrency, and threads are currently the dominant interface for expressing concurrency in imperative languages. Therefore, part of this

project is the implementation of an application thread library that utilizes the scheduler activations interface. The library is intended to become the supported POSIX-compatible ("pthreads") library for NetBSD.

The thread library uses the scheduler activations interface described previously to implement POSIX threads. The threads are completely preemptable; the kernel may interrupt a thread and transfer control to the upcall handler at any time. In practice, this occurs most often when a system call blocks a thread or returns from having been blocked, or after another process on the system has preempted the threaded process and then returned. On a uniprocessor system, for example, logging in remotely via `ssh` and running a threaded process that produces terminal output causes frequent preemptions of the threaded process, as the kernel frequently allocates time to the `sshd` process to send terminal output back to the user. Periodic timers are used to generate regular upcalls to implement round-robin scheduling, if requested by the application.

The complete preemptability of scheduler activations threads is a problem in that it violates the atomicity of critical sections of code. For example, the thread library maintains a run queue; if an upcall happens while the run queue is being operated on, havoc will result, as the upcall handler will itself want to manipulate the run queue, but it will be in an invalid state. When spin locks are used to protect critical sections, this problem can lead to deadlock, if the upcall handler attempts to acquire the same lock that was in use by the interrupted thread.

Both the original scheduler activations work and the Mach implementation faced this problem, and both adopted a strategy of recovery, rather than prevention. That is, rather than violate the semantics of scheduler activations by providing a mechanism to prevent interruption during a critical section, they devised ways to detect when a critical section had been interrupted, and recover from that situation.⁴ In both implementations, the upcall handler examines the state of each interrupted thread to determine if it was running in a critical section. Each such thread is permitted to run its critical section to completion before the upcall handler enters any critical section of its own.

The implementation of critical-section recovery in this thread library closely follows the Mach implementation. Critical sections are protected with spin locks, and

⁴Adding to the terminological confusion, some Sun Solaris documentation refers to the `schedctl_start()` and `schedctl_stop()` functions, which temporarily inhibit preemption of a LWP, as "scheduler activations".

the spin lock acquisition routine increments a spin lock counter in the acquiring thread's descriptor. When an upcall occurs, the upcall handler checks the spin lock count of all interrupted threads. If it finds an interrupted thread that holds spin locks, it sets a flag in the descriptor indicating that the thread is being continued to finish its critical section, and then switches into the context of that thread. When the critical section finishes, the spin lock release routine sees the continuation flag set in the descriptor and context switches back to the upcall handler, which can then proceed, knowing that no critical sections are left unfinished. This mechanism also continues the execution of any upcalls that are themselves preempted, and that may have been continuing other preempted critical sections.

While this system is effective in preventing problems with preempted critical sections, the need to manipulate and examine the thread's descriptor in every spin lock operation undesirably adds overhead in the common case, where a critical section is not preempted. An area for future exploration would be a mechanism more similar to that of Anderson's original implementation, which uses knowledge of which program addresses are critical sections to shift all of the costs of preemption recovery to the uncommon case of a critical section being preempted.

The thread implementation also has a machine-dependent component, although it is much smaller than the kernel component. Short routines that switch from one thread to another (saving and restoring necessary register context), and variants of these routines that are needed for the preemption detection described above, are needed for each CPU type supported by NetBSD.

7 Performance Analysis

There are two goals of examining the performance of the scheduler activations system. The first is determining whether the added complexity of having scheduler activations in the kernel hurts the performance of ordinary applications. The second is comparing the performance of the resulting thread system with existing thread systems to demonstrate the merits of the scheduler activations approach.

The first measurements were done with the HBench-OS package from Harvard University [3]. HBench-OS focuses on getting many individual measurements to explore the performance of a system in detail. Five of the

tests measure system call latency; the results from the NetBSD kernel before and after the implementation of scheduler activations are shown here, as measured on a 500 MHz Digital Alpha 21164 system.

	before SA	after SA
getpid	0.631	0.601
getrusage	4.053	4.332
timeofday	1.627	1.911
sbrk	0.722	0.687
sigaction	1.345	1.315

The results are mixed; some tests are faster than before; others are slower. The larger set of HBench-OS tests test process-switch latency for a variety of process memory footprints and number of processes being switched; that set showed similarly mixed results. So while the performance of the system changed slightly, it did not conclusively get faster or slower. This result makes the work as a whole more attractive to the NetBSD community, since it does not require accepting a performance trade-off in order to get a better thread system.

For measuring thread operation costs, three different micro-operations were measured: The time to create, start, and destroy a thread that does nothing; the time to lock and unlock a mutex (under varying degrees of contention), and the time to switch context between exiting threads.

The operations were measured on an Apple iBook, with a 500MHz G3 processor with 256k of L2 cache. The tests were conducted with the scheduler activations thread library on NetBSD, the GNU PTH library on NetBSD, and LinuxThreads on Linuxppc 2.4.19.

	SA	PTH	Linux
Thread	15 μ s	96 μ s	90 μ s
Mutex	0.4 μ s	0.3 μ s	0.6 μ s
Context	225 μ s	166 μ s	82 μ s

LinuxThreads exhibited roughly linear scaling of lock time with the number of threads contending for the lock; neither the SA library nor PTH had noticeable increases in lock time, demonstrating interesting differences in the scheduler involved. The PTH null thread creation time is also surprisingly large, for a pure user-space thread library that should have low overhead. Linux does quite well at the context-switch test, and it will be worth investigating how to get that speed in NetBSD.

In all three benchmarks, The scheduler activations threads demonstrated that basic operations are competitive with both pure user threads and 1:1 kernel threads.

8 Conclusions and future work

This paper has presented the design and implementation of a two-level thread scheduling system based on the scheduler activations model, including the kernel interface, kernel implementation, and user implementation. Measurements were taken that demonstrate both competitive thread performance and no sacrifice in non-threaded application performance. The implementation is sufficiently well-divided between machine-dependent and machine-independent parts that porting to another architecture is only a few days' work. As was initially hypothesized, scheduler activations is a viable model for a thread system for NetBSD. The existence of a scheduler activations implementation in a portable, open source operating system will enable further research into the properties of this appealing system.

This project continues to evolve, but several future goals are clear: integration with the main NetBSD source tree; cooperation with the fledgling support for symmetric multiprocessing in NetBSD; implementation of better critical section preemption as described, implementation of optional POSIX threads features such as realtime scheduling, and as always, performance tuning.

Availability

The kernel and user code described here is available under a BSD license from the NetBSD Project's source servers⁵, currently on the CVS branch called `nathanw_sa`. Machine-dependent code has been written for the Alpha, ARM, i386, MIPS, Motorola 68k, PowerPC, and VAX architectures, with more on the way. Integration into the trunk of NetBSD-current is expected after the release of NetBSD 1.6.

Acknowledgments

Thanks to Frans Kaashoek of the MIT Laboratory for Computer Science for supervising early stages of this project, and to Wasabi Systems for sponsoring continued development. Thanks to Klaus Klein for providing a Single Unix Specification-compliant implementation of `ucontext_t` and its associated system calls. Thanks

⁵Please see <http://www.netbsd.org/Documentation/current/> for ways of getting NetBSD

to Bill Sommerfeld and Jason Thorpe for review of the interface and many useful suggestions throughout the work on this project. Finally, thanks to Chris Small and Margo Seltzer for their implementation of scheduler activations on BSD/OS; I am indebted to them for their demonstration of feasibility.

References

- [1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. 19th ACM Symposium on Operating System Principles*, pages 95–109, 1991.
- [2] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska. Adding scheduler activations to Mach 3.0. Technical Report 3, Department of Computer Science and Engineering, University of Washington, August 1992.
- [3] A. Brown and M. Seltzer. Operating system benchmarking in the wake of Imbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, 1997.
- [4] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN 0-201-63392-2.
- [5] Digital Equipment Corporation. *Guide to DECthreads*. Digital Equipment Corporation, 1996. Part number AA-Q2DPD-TK.
- [6] Ralf S. Engelschall. Gnu portable threads. <http://www.gnu.org/software/pth/pth.html>.
- [7] Portable threads library. <http://www.media.osaka-cu.ac.jp/~k-abe/PTL/>.
- [8] Christopher Provenzano. Portable thread library. <ftp://sipb.mit.edu/pub/pthreads/>.
- [9] Christopher Small and Margo Seltzer. Scheduler activations on BSD: Sharing thread management between kernel and application. Technical Report 31, Harvard University, 1995.

Authorization and charging in public WLANs using FreeBSD and 802.1x

Pekka Nikander

*Ericsson Research NomadicLab
pekka.nikander@ericsson.com*

Abstract

The IEEE 802.1x standard defines a link-layer level authentication protocol for local area networks. While originally designed to authenticate users in a switched Ethernet environment, it looks like the most important need for 802.1x lies in wireless networks, especially IEEE 802.11b based Wireless LANs. Furthermore, due to the flexibility of the Extensible Authentication Protocol (EAP), the heart of 802.1x, it looks like 802.1x could be used for many purposes its original designers have not foreseen.

In this paper, we describe an FreeBSD-based open source 802.1x implementation, and show how it can be used to implement different authorization and charging systems for public WLANs, including a pre-paid, pay-per-use charging system and another one based on community membership. The implementation is based on the netgraph facility, resulting in a surprisingly flexible and simple implementation.

1 Introduction

With the advent of 802.11 based Wireless Local Area Networks (WLANs) and the proliferation of laptops and PDAs, the usage of Ethernet networks is changing. What once was meant to statically connect computers belonging to a single organization, is now increasingly being used to provide Internet connectivity for mobile professionals. This change is very visible at certain high profile telecommunication and computer science conferences, including IETF meetings, where an increasing number of people are utilizing the wireless networks provided by the organizers.

The changing usage patterns brings forth new security problems. From the network point of view, the most prominent problem is that of access control. The network must somehow decide if and how to allow a network node to utilize the resources provided by the network. The IEEE 802.1x standard [1] is designed to provide a solution framework for this problem.

The 802.1x standard defines a method to run the Extensible Authentication Protocol (EAP) [2] in raw (non-IP) Ethernet frames. The resulting protocol is called EAP over LAN (EAPOL). The EAP protocol was originally designed as a flexible authentication solution for modem connections using the Point-to-Point

Protocol (PPP). Now 802.1x is extending the same architecture to LANs, both wireline and wireless.

At the basic level, the 802.1x architecture, properly augmented with the RADIUS [3], allows the existing PPP based authentication, authorization and accounting infrastructure to be used to control LAN access in addition to PPP access. The solution works well in current environments, where the network providers and network users have an existing, pre-established relationship. However, if we consider public WLAN offerings at airports, hotels, cafes, and even offices and homes, it becomes more common than before that the prospective network user has no relationship with the provider. Thus, something new must be introduced, something that allows such a relationship to be build on the fly.

In this paper, we show how it is possible to set up a public WLAN environment that supports various kinds of authorization and charging schemes, including community membership based and pre-paid, pay-per-use accounts. The new schemes allow more flexible user-provider relationship management than the existing ones. The implementation is based on IEEE 802.1x, and the prototype runs under FreeBSD. The architecture is geared to small service providers, eventually scaling down to the level of individuals offering WLAN based Internet access through their xDSL or cable modem.

The rest of this paper is organized as follows. In Section 2, we briefly describe the IEEE 802.1x standard and the FreeBSD netgraph facility. After that, in Section 3, we describe our 802.1x implementation in broad terms, and in Section 4 we cover the implementation details. In Section 5 we describe how the implementation can be configured to support different usage scenarios, including pre-paid pay-per-usage accounts and community membership based accounts, and Section 6 includes our initial performance measurements. In Section 7, we discuss a number of open issues and future possibilities related to this ongoing work. Finally, Section 8 includes our conclusions that we have been able to achieve so far.

2 Background

Our work is mostly based on existing technologies, integrating them in a novel way. The only significant piece of new software that we have written is the base 802.1x EAPOL protocol [1], which is implemented as a pair of FreeBSD netgraph [4] nodes. Since we cannot assume that all readers are familiar with the IEEE 802.1x standard and the netgraph facility, we introduce them briefly in this section. Additionally, we discuss related work.

2.1 IEEE 802.1x

IEEE 802.1x [1] is a forthcoming standard for authenticating and authorizing users in Ethernet like local area network (LAN) environments. It is primarily meant to secure switched Ethernet wireline networks and IEEE 802.11 based WLANs. In the typical usage scenarios, the network requires user authentication before any other traffic is allowed, i.e., even before the client computer is assigned an IP address. This allows corporations to strictly control access to their networks.

The 802.1x overall architecture is depicted in Figure 1. The central point of the architecture is an Ethernet switch or WLAN access point. Now, instead of directly connecting clients to the network, the access points contains additional controls, basically preventing the clients from communicating before they have positively authenticated themselves. The authentication takes place between an EAPOL supplicant, running on the client, and a background authenticator server. This is denoted with the thick, dashed arrow line. The authentication protocol can be any supported by the Extensible Authentication Protocol (EAP) standard [2].

Even though the actual authentication protocol is run between the supplicant client and the authentication

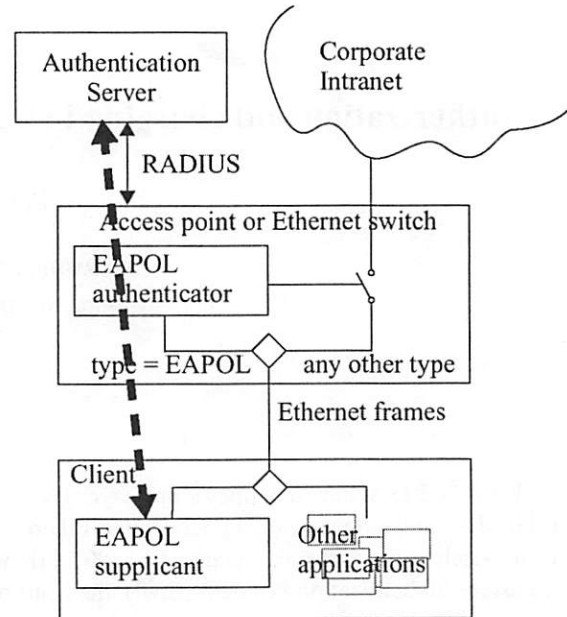


Figure 1: Basic IEEE 802.1x architecture

server, the protocol is mediated by an EAPOL authenticator function located at the access point. The EAPOL authenticator takes care of two functions. First, it mediates the EAP packets between RADIUS, used towards the authentication server, and EAPOL, used towards the client. Second, it contains a state machine that “snoops” the EAP protocol, learning whether the authentication server was able to authenticate the user identity or not. If the user was authenticated, it permits the client to freely communicate; otherwise the client is denied access from the network. In the figure, this latter function is depicted with the on/off switch within the access point.

It is important to note that 802.1x implements only authentication and MAC address based access control. Since MAC spoofing is fairly easy, the resulting system, as such, might not be secure enough. The need for additional security measures depends on the usage scenario; see Section 7.1 for the details.

2.2 EAPOL

The EAPOL protocol is run between the EAPOL supplicant, running on the client, and the EAPOL authenticator, running on the access point. It is a fairly simple protocol, consisting of a packet structure and state machines running on both the supplicant and the authenticator. The packets are based on raw Ethernet frames with little additional structure, as depicted in Figure 2, on the next page.

The protocol is typically initiated by the authenticator as soon as it detects that a new client has been connected

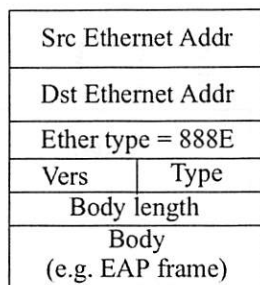


Figure 2: EAPOL packet structure

to an switched Ethernet port or that a new client has joined to a WLAN access point. However, the protocol can also be triggered by the client by sending an EAPOL start packet.

A typical protocol run is depicted in Figure 3, below. The first message is an EAP identity request, sent by the EAPOL authenticator. The supplicant replies with an EAP identity response, which is passed to the authenticator server. The authenticator server replies by running one of the possible EAP subprotocols within EAP request and response frames. Once the authentication phase has been completed, the authentication server sends an EAP success (or failure), indicating that the user identity was verified (or could not be verified). The EAPOL authenticator notices this message, and allows (or disallows) the client to communicate.

In addition to the base authentication, EAPOL also allows periodic re-authentication and logout by the client. These are most useful in a setting where connection time is used as a basis for accounting.

2.3 Netgraph

Netgraph [4] is a flexible network protocol architecture implemented in the FreeBSD kernel. Currently, it allows different link-layer protocols to be stacked between the device drivers and the network layer protocols, i.e., IP. Its basic benefit is flexibility; it makes it easy to add new link-layer subprotocols, and to stack different protocols,

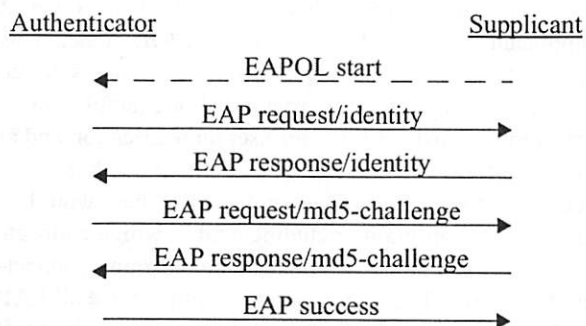


Figure 3: A typical EAPOL protocol run

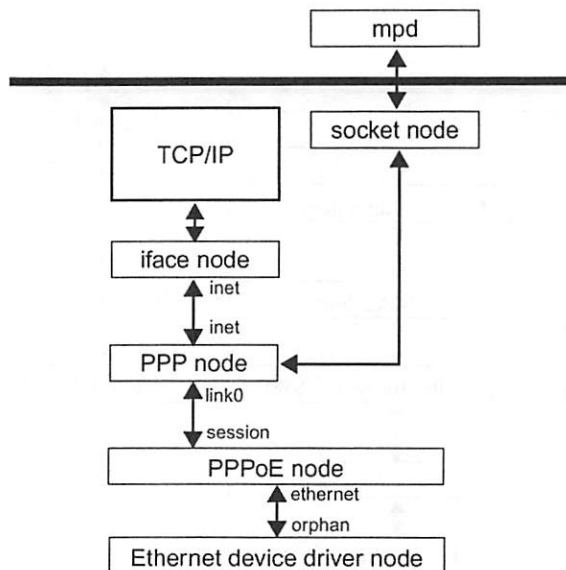


Figure 4: A netgraph stack implementing PPPoE

such as HDLC, Frame Relay, and PPP, in different ways. Furthermore, it allows intelligent filtering and pseudo-interfaces, thereby making it possible to make a difference between LAN clients based on their MAC addresses.

From the implementor's and administrator's point of view, netgraph is extremely flexible. Firstly, new protocol nodes are implemented as loadable kernel modules, making development and testing fairly easy. Secondly, the stacking order and connections between the protocol nodes are configured with a user level program. This makes it possible to use the same protocol nodes in many different ways. (See the examples in Section 5.)

A simple netgraph stack is shown in Figure 4. The stack is used to run PPP over Ethernet (PPPoE), a networking standard that a number of xDSL and cable operators use today. The actual PPP signalling is performed at the user level, using a separate daemon (mpd). However, the data packets flowing between the IP protocol and the physical interface are completely handled at the kernel level.

2.4 Related work

Even though there are a number of commercial 802.1x implementations available, the only other open source implementation that the author knows of is one by the Open1x project at University of Maryland, College Park [5]. Compared with our project, there are two major differences. Firstly, Open1x is a Linux based user level implementation while ours relies on netgraph. Secondly, based on the first source code release, the Open1x project seems to be more research oriented, trying to

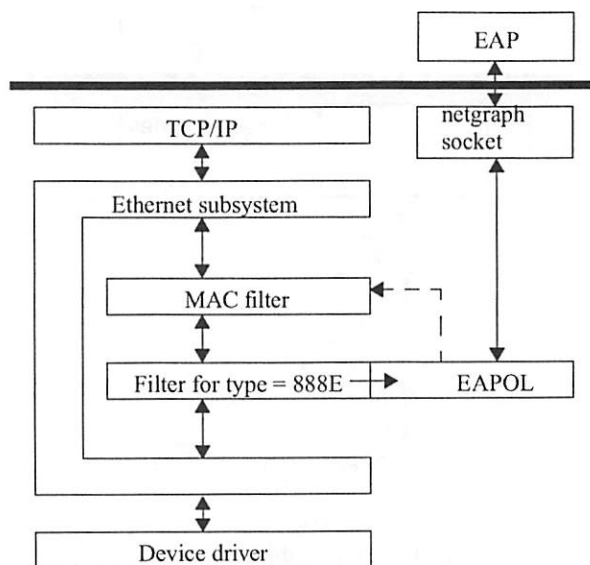


Figure 5: A netgraph stack implementing 802.1x compliant access point

identify security problems in the specifications, while our goal has been to produce a high quality implementation that could be used in production environments.

3 Software architecture

In this section, we describe our solution architecture in broad terms. The details are left for the next section, and a couple of usage scenarios are given in Section 5. In this section, we first describe the overall structure of the software, and then briefly describe the components.

3.1 Overall structure

Our 802.1x implementation consist of two new netgraph nodes, plus a number of user level programs. The user level programs implement the individual EAP subprotocols. The larger of the netgraph nodes implements the EAPOL protocol, providing a clean interface to the user level EAP programs, while the smaller netgraph node implements a simple MAC address filter.

A basic netgraph structure, used to implement a simple 802.1x compliant access point, is depicted in Figure 5, above. In this configuration, the 802.1x implementation is hooked inside the Ethernet subsystem (`if_ethersubr.c`) using suitable negraph hooks.

If we consider incoming packets, they are first routed to the first part of the EAPOL module. If the ethertype is `0x888E` (EAPOL) the packets are passed to the second part of the EAPOL node. Otherwise they are passed

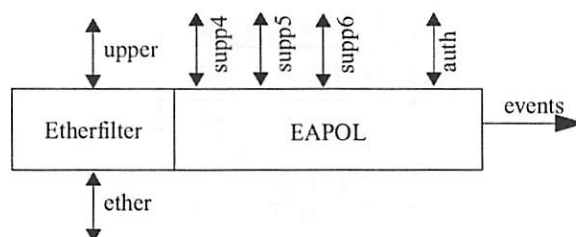


Figure 6: EAPOL node hooks

upward to the MAC filter. The MAC filter either passes or drops the packet, depending on the sender's MAC address. The list of passed addresses is maintained by the EAPOL module by sending events to the MAC filter. The passed packets are simply routed back to the Ethernet subsystem, which then passes the packets to the upper layer protocol.

3.2 EAPOL netgraph node

The EAPOL netgraph node implements the EAPOL protocol at the kernel level. It includes both EAPOL authenticator and supplicant functionality, and is able to run both functions in parallel. However, it handles directly only the EAP identity and notification request/response exchanges. All actual authentication protocols must be implemented outside the EAPOL node. The node just includes hooks that different EAP protocol implementations can attach to. Due to the flexibility of netgraph, the protocols can either run inside the kernel, as netgraph nodes, or at the user level.

The external interfaces of the EAPOL node are described in Figure 6, above. The node filters received Ethernet frames and passes non-EAPOL frames to upper hook. Any packets received at the upper hook are passed to the ether hook. The events hook reports EAPOL related events, such as EAPOL start and logoff frames and EAP success and failure packets.

The `supp4..suppN` hooks are used to connect supplicant EAP modules to the EAPOL node. This architecture allows different EAP subprotocols to be implemented by different programs. For example, if one wants to use EAP TLS [6] for user authentication and at the same time EAP OTP [2][7] for client machine level access control, it is possible. In that case, there would be an EAP TLS program, including a GUI, which connects to `supp7`, and an EAP OTP daemon program, connecting to `supp5`. The TLS program would receive all EAP packets containing TLS related requests, and the OTP program those containing OTP related requests.

We originally planned to have a similar structure at the authenticator side, too. However, it turned out to be hard to keep track of sent requests and received responses so that the response packets could be delivered to the right EAP authenticator. The reason for this is that the tracking is based on the 8 bit EAP message ID code in the requests and matching responses. To support several parallel authenticators, it would be necessary to select unique message IDs to the outgoing requests, and to replace the message IDs with the original ones as responses are received. Since this functionality is usually not needed, the added complexity and kernel level state just seemed unnecessary.

Consequently, the author revised the base EAPOL module so that it supports only one authenticator hook and delivers all received EAP response packets to this hook. This serves very well integrated authentication systems where there is a back-end server. It is still possible to reintroduce the original structure by implementing an additional netgraph node that keeps track of outstanding requests and performs the necessary ID conversions.

3.3 User level EAP programs

As already discussed, the architecture allows the different EAP authentication protocols to be implemented as distinct user level programs. At the client (supplicant) side, the program simply receives EAP requests from the EAPOL netgraph node, processes them, and sends back EAP responses. If desired, the program can easily implement a user interface to request information from the user.

At the server (authenticator) side the situation is slightly more complex. Firstly, the authenticator must keep track of outstanding requests, based on the message IDs. Secondly, a single program usually wants to support several different authentication protocols, for example, since it wants to connect to a back-end authentication server. Furthermore, the program may need additional information about the client, such as the MAC address and the physical interface through which the packet was received.

The first two needs are covered by the prototype system. However, we do not currently pass information about the interface, nor the MAC address, to the user level. In the simple scenarios it is not needed, since the required information is passed directly within the kernel between the modules. On the other hand, we are plan-

ning to modify `ng_socket` so that it is able to pass metadata to the user level and vice versa. Once that facility is available, it becomes possible to pass the MAC address and interface index in `sockaddr_dl`. That would even allow future interoperability with PPP EAP.

In the simplest case, a server side EAP program simply receives EAP responses from the EAPOL netgraph node, converts them into RADIUS requests, and sends them to the RADIUS server. Similarly, it receives EAP requests within RADIUS packets from the RADIUS server, converts them back, and sends down to the EAPOL module. The EAP program does not need to take care of retransmissions, since the underlying EAPOL state machine takes care of that.

An alternative design would be to use locally authentication data, such as passwords or OTP keys. In that case, the authenticator module would receive an EAP identity response, look up the user from the local authentication database, create state, send an appropriate EAP request, and wait for a response. Once it receives a matching response, it would compare the received authenticator against the expected one, and either pass or reject the user. For an example of such a module, see Section 4.3.

3.4 MAC filter netgraph node

In addition to supporting the EAPOL protocol, an 802.1x authenticator also needs to include the functionality that blocks unauthenticated clients from communicating with the network behind the access point. To support this, we have implemented another netgraph node, the MAC filter. The MAC filter allows incoming packets to be routed to different hooks depending on the source MAC address. If there is no matching hook, the packets are passed to a default hook. If no other node is connected to the default hook, the packets are simply dropped.

In addition to allowing packets to be screened by their source MAC address, as required by base 802.1x, the design also supports more sophisticated modes of operation. Basically, different clients can be classified in different categories, and handled appropriately. This allows different service classes for different packets and direct bridging some packets to an Ethernet-level tunnel while other packets are allowed to reach the local IP routing layer. One possible scenario, multioperator support, is described in briefly in Section 5.3.

4 FreeBSD implementation

In this section we describe the implementation in detail. Due to the space available, we mainly describe the design related to the netgraph facility. The rest of the implementation is straightforward and does not really need much comment at the implementation level. In this section, all reported object code figures refer to code compiled in FreeBSD 4.5-STABLE on i386, using the supplied GNU gcc 2.95.3 tool chain.

4.1 EAPOL netgraph node

The EAPOL netgraph node is a new netgraph node that implements the EAPOL protocol as defined in IEEE 802.1x draft 11 [1]. However, since EAPOL includes a few layering violations, the EAPOL node is forced to have some rudimentary understanding of the protocol above, EAP, as well. That is, an EAPOL implementation is expected to send a few canned EAP packets (identity request, forced success, and forced failure) as well as understand the meaning of success and failure packets as received from an authentication server.

File name	Source lines	Object code size
ng_eapol.c	1038	3980 + 164
ng_eapol_base.c	255	1332 + 68
ng_eapol_auth.c	658	2756 + 508
ng_eapol_supp.c	552	2432 + 372
ng_eapol_macfilter.c	89	372 + 0
ng_eapol.h	77	
ng_eapol_base.h	326	
Total ng_eapol.ko	2995	19013 + 1220

Table 1: The EAPOL netgraph module

The implementation is divided into four source files (see Table 1). `ng_eapol.c` implements the netgraph related code, `ng_eapol_base.c` implements functionality common to both authenticator and supplicant, while `ng_eapol_auth.c` and `ng_eapol_supp.c` the role specific functionality. We also considered separating the authenticator and supplicant functionality into separate netgraph modules, but given that over one third of the code volume is spent on netgraph glue, we decided to integrate them into a single node instead. However, it is still possible to leave out either authenticator or supplicant functionality (or both) through compile-time options. This also simplifies testing.

```
handle_eapol(...,
    struct mbuf *m,
    meta_p meta) {
    struct eapol_hdr *ep = mtod(...);

    switch (ep->eapol_code) {
    case EAPOL_CODE_EAP:
        return handle_eap(..., m, meta);
    case EAPOL_CODE_LOGOFF:
        eapolp->eapol_state = LOGOFF;
        NG_FREE_M(m);
        NG_FREE_META(meta);
        return 0;
    ...
    }
```

Figure 7: Result of functional programming style

Another design choice we faced was between a functional vs. procedural programming style. In a more functional style, the program is divided into separate functions that have little if any side effects. Each piece of code performs memory management separately. In a pure functional programming language, such as Lisp or ML, memory management is taken care by the runtime, and side effects are virtually nonexistent. In a more procedural style, procedures act on data structures, and memory management centers around the them.

In our case, each netgraph node is separately responsible of either passing all packets received to another node, or explicitly freeing the mbuf and meta objects that comprise the packet. In the first version of the code, the actual code for these operations was distributed to the point of code where the decision was made. A simplified example is shown in Figure 7, above. This led to a situation where it was fairly hard to ensure that there was no memory leaks, since a single function was forced to free the mbuf and meta sometimes while sometimes delegating this to the called functions.

After realizing this, the code was refactored leading to a design where the packet is either forwarded or freed centrally at the very first function where the packet enters the node, `ng_eapol_rcvdata`. This makes sure that there are no memory leaks, but this also necessitates that, instead of passing mbufs, callers pass a pointer to the original mbuf pointer, and an output parameter that specifies a netgraph hook. If the returned hook is non-NULL, the packet is passed to the hook returned. The resulting code is depicted in Figure 8, on the next page.

In the refactored code, the metadata does not need to be passed around any more (unless it is used, or course),


```

handle_eapol(...,
    struct mbuf **mp,
    hook_p *nhook) {
    struct eapol_hdr *ep = mtod(...);

    switch (ep->eapol_code) {
    case EAPOL_CODE_EAP:
        return handle_eap(..., mp, nhook);
    case EAPOL_CODE_LOGOFF:
        eapolp->eapol_state = LOGOFF;
        *next_hook = NULL;
        return 0;
    ...
    }
}

```

Figure 8: Refactored code

and freeing the packet is replaced by setting the next hook output parameter `nhook` to `NULL`, thereby signaling that the packet should be freed and not passed on. Passing `mbuf **` instead of `mbuf *` is necessitated by some mbuf routines explicitly freeing the mbuf in the case of mbuf shortage. By passing a pointer to the mbuf pointer, we avoid freeing mbufs twice in those cases.

Thus, the basic lesson learned was that the functional programming style that the author is used to use with languages that include garbage collection just does not work when you have to take care of memory management yourself. It is much better to centralize the actual code that manages memory, and use extra parameters to signal the decision of what to do from the actual decision point to the centralized piece of code.

Other than the issues with granularity and programming style, implementing the EAPOL node was pretty straightforward translation of the standard specification into working code.

4.2 MAC filter netgraph node

The MAC filter node, `ng_macfilter`, is a fairly simple mux/demux. It is build around an ordered table of MAC addresses. Each address in the table is annotated with the index of an upper netgraph hook. All packets

File name	Source lines	Object code size
<code>ng_macfilter.c</code>	792	2032 + 100
<code>ng_macfilter.h</code>	60	
Total <code>ng_macfilter.ko</code>	852	4110 + 200

Table 2: The macfilter module

received from any of the upper hooks are passed directly to the lower hook. On the other hand, whenever a packet is received from the lower hook, the source MAC address is inspected to see if it matches with any of the addresses in the table. If a match is found, the packet is passed to the indicated upper hook. If no match is found, the packet is passed to the default upper hook.

4.3 User level modules

To test the netgraph modules and to implement one of our usage scenarios, we implemented EAP OTP authenticator and supplicant modules. These modules both work on the user level, and rely on a small new library, `libeap`. They also utilize the `libskey` library present in FreeBSD.

File name	Source lines	Object code size
<code>libeap/eap_auth.c</code>	144	1020 + 0 + 0
<code>libeap/eap_peer.c</code>	165	788 + 0 + 0
<code>eap-opie/opie.h</code>	25	
<code>eap-opie/opie_calc.c</code>	108	436 + 48 + 228
<code>eap-opie/opie_auth.c</code>	149	940 + 0 + 1536
<code>eap-opie/opie_peer.c</code>	112	764 + 0 + 1536
<code>opie_auth binary</code>	10212 bytes	5125 + 380 + 2348
<code>opie_peer binary</code>	9575 bytes	4553 + 368 + 2116
Total source code	703 lines	

Table 3: The user level modules

As Table 3 shows, the actual sizes of the user level programs, as reported by `size(1)`, are extremely small.

4.4 Overall experiences with netgraph

This project was the first time when the author used netgraph, even though he had some prior experience in working with kernel level code. In general, netgraph turned out to be a very well designed facility that boosted the project productivity considerably. The main benefit was the no-hassle startup, and the ability to write all the code as loadable kernel modules. That is, once the author decided to use netgraph, the first null version of the to-be EAPOL module was loaded into the kernel within a couple of hours, most time spent on reading netgraph documentation. The documentation was clean and concise; the only part that the author found lacking was the usage examples, which were somewhat hard to understand. That led to some unnecessary work before

really understanding how to use the `ngctl` program and how the generic `netgraph` command messages such as `mkpeer` or `connect` work. In general, it looks like the `netgraph` facility provides an extremely good environment for implementing and connecting together sub-IP protocols in FreeBSD.

5 Usage scenarios

In this section, we describe three different advanced usage scenarios. These all go beyond those intended to be implemented by the base IEEE 802.1x standard, which is mainly meant to protect corporate intranets from outsiders. In the first scenario, we outline a community membership based authorization model, where members of a community may use the network regardless of their identity. In the second scenario we outline how to implement a pre-paid, pay-per-use scenario suitable for early public WLAN adopters at airports and hotels. Finally, in the last scenario we outline how to introduce multioperator support into a WLAN access network.

The different scenarios have different threat and trust models. We discuss the security needs of the scenarios only briefly; more thorough treatment of the underlying trust models is beyond the scope of this paper.

None of these scenarios have been fully implemented. The second, pre-paid, pay-per use scenario is close to being really implemented, but even there are bits and pieces that are missing before it could be used in real life. The scenarios are provided to exemplify how the flexibility of the implementation allows it to be used as a building block in fulfilling different needs.

5.1 Community membership

There are number of open WLAN communities, e.g. Seattle Wireless [8] or Electrosmog [9], where the members of the communities are basically providing free WLAN based Internet access to anybody passing by. While these schemes work fine today, they are likely to suffer from “the tragedy of the commons” phenomenon as the number of WLAN users starts to grow. One possible way to continue the spirit of these networks while protecting them from overuse is to limit their usage to members, i.e., to create closed communities.

Using 802.1x, EAP TLS [6], and certificates, it seems to be fairly easy to create closed but decentralized communities. Let us assume that there are a number of founding members that create the community, and that any two of the founding members can accept new members to the community. One way of implementing such a

scheme is to use authorization certificates [10], and to let all the founding members to sign a policy certificate stating that any two members are eligible to introduce new members. Thus, instead of having a single centralized certification authority (CA) the community would distribute the responsibility among the founding members.

All community access points would be connected to a centralized repository that contains the initial policy certificates. When new users come to an access point, their laptop would run EAP TLS and send the two certificates signed by the founding members to the access point. The access point would fetch the policy certificates corresponding to the signers of the user certificates, and make the access decision normally using the KeyNote2 engine.

In this way, the community would limit access to its members, while still retaining the decentralized nature of deciding who is a member and who is not. This solution requires that KeyNote2 [10] is integrated to OpenSSL [11], and that EAP TLS is created using the OpenSSL library.

Compared to NoCatAuth [12][13], probably the best known open source effort on this area, this 802.1x based solution would be better in two ways. Firstly, the 802.1x allows the authentication to happen without any user intervention. Secondly, the use of authorization certificates allows the administration of group membership to be completely decentralized.

5.2 Pre-paid, pay-per-use

As another scenario, we show how OTP [7] accounts can be used as pre-paid pay-per-use value tokens, and how it is possible to easily add a captive portal to the access point. Integrating these with some kind of on-line macropayment system, such as credit card payment, it is possible to create a pre-paid, pay-per-use public WLAN that is open to any users.

The first key element in this scenario is to realize the potential of OTP accounts and re-authentication. That is, the periodic re-authentication feature of 802.1x allows the access point to request the next OTP token from the user. Thus, we can establish a convention where a single OTP token represents the value for a certain commodity, e.g., for one minute of access time. Once the commodity has been used up, the access point can request for the next token in order to continue the service.

In this way, an OTP account can be used to represent a pre-paid, pay-per-use account. The user buys such an account with a credit card or some other means, and gets

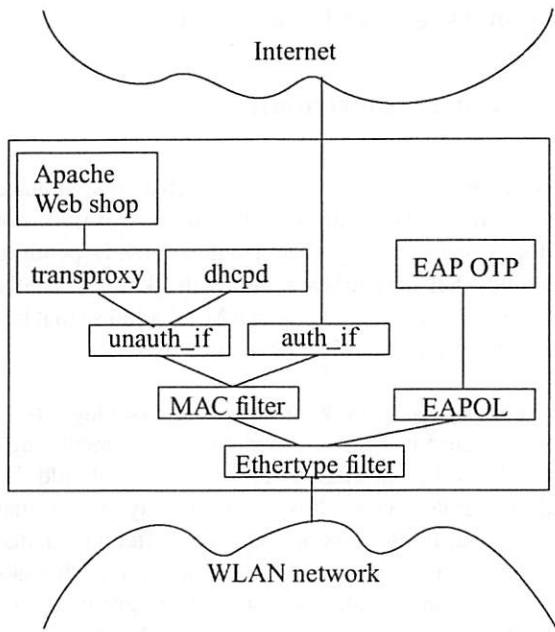


Figure 9: Pre-paid, pay-per-use access point

the account name and the OTP seed value. In most cases there is no value in requesting a password from the user, and therefore it is possible to automate the whole purchase process. For example, a small web browser plugin can be used to transfer the account from the shop to a preset location at the user's hard disk. The EAP OTP supplicant is then able to select the next unused account from this location.

The final step is to make it easy to buy new OTP accounts. It must also be easy to become a user, i.e. to buy the initial OTP account and, if necessary, download the 802.1x and EAP OTP implementations. This can be easily accomplished with a captive portal. As we described in Section 3.4, our MAC filter module allows packets arriving from unauthenticated users to be passed on instead of being dropped. The netgraph facility allows these packets to be passed to an pseudo-interface, which can then pass them to IP. Using the FreeBSD IP firewall, ipfw, it is then easy to trap HTTP all packets arriving from the specific pseudo-interface, and use a forwarding rule (fwd) to pass them to a transparent web proxy such as transproxy [14]. The transparent proxy can then tunnel the packets to a web server, which can be located either at the access point itself or somewhere else.

Thus, with suitable configuration it is possible to create a situation where the only services provided to an unauthenticated client are DHCP and the captive portal. This effectively leads to a situation where the client is served an IP address, but no matter which web site the users attempt to access, they will be redirected to the

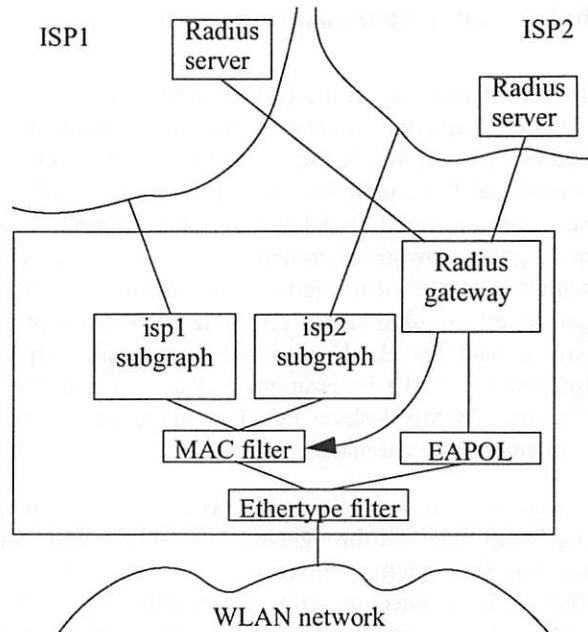


Figure 10: Multioperator access point

captive portal. The captive portal web site, in turn, provides the possibility to buy OTP accounts and to download all necessary software.

5.3 Multioperator support

As a final usage example, we show how the extensibility of our implementation can be used to support a multioperator scenario where a single WLAN network is shared between multiple ISPs and other service providers. In this case, we need a slightly more intelligent EAP authenticator. This authenticator needs to be able to talk to multiple back-end authentication servers — RADIUS seems to be fine for that — and, more importantly, to command the MAC filter to connect the clients to different upstream netgraph hooks depending on the commands received from the authentication servers. A possible setup is depicted in Figure 10.

The upstream hooks are all individually connected to a subgraph that transports the packets to the ISP. In a very simple (but unrealistic) example, each ISP would have a separate Ethernet interface in the access point. In that case, the hooks are simply connected to the Ethernet interface, and the access point just bridges the packets, selecting the outgoing interface based on the source MAC address. In a more realistic case, there would be some netgraph subgraph that tunnels the packets to the ISP, using e.g. 802.1q VLAN or PPPoE. Netgraph allows one to build such subgraphs fairly easily. As a result, all the data packets would be handled inside the kernel, potentially leading to fairly good performance.

6 Performance measurements

To get a feeling about the real life performance of the technology and implementation, we implemented a limited pay-per-use WLAN scenario. In the setting we assumed that the laptop user already has purchased the necessary tokens (the OTP account) and is merely connecting to a network where the tokens can be used. The setting is illustrated in Figure 11, below. In the performance test, we used an old 400 MHz PII 64 Mb laptop, with Lucent WaveLAN silver card, an Apple Airport (graphite) 802.11b basestation, and an old 350 MHz Celeron 128 Mb desktop PC. The Apple Airport was configured to be a transparent bridge.

Using the given configuration, we measured connection setup time and throughput, both without and with our 802.1x implementation. The results are given in Table 4. The connection setup measures the time, in seconds, from inserting the Lucent WLAN card to the laptop to the time when the laptop received the IP address via DHCP. The bulk transfer throughput measures the additional overhead caused by the MAC filter in the kernel, using FTP file transfer as the measurement tool. The values are averages over five measurements.

Measurement	Without 802.1x	With 802.1x
Connection setup	22 \pm 2 s	24 \pm 3 s
Bulk transfer throughput	830 KB/s	820 KB/s

Table 4: Performance results

The results show that the overhead is negligible. The connection setup takes 2–3 seconds longer, an increase of roughly 10%. This is consistent with our other measurements that showed the 802.1x exchange itself taking 1.1 ± 0.4 seconds on the average. The performance tests gave slightly better performance when using 802.1x, possibly due to interactions with 802.3 or 802.11b queuing and retransmission algorithms.

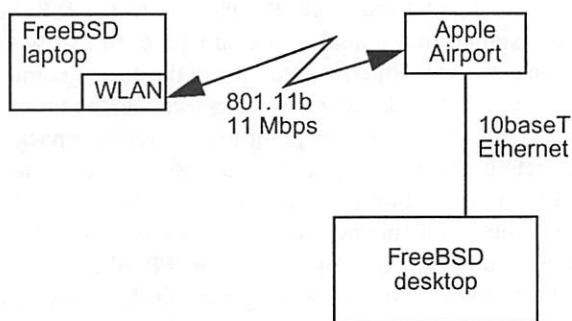


Figure 11: Performance Measurement Setting

7 Open issues and future work

7.1 Security shortcomings

Many people, including Mishra and Arbaugh [15], have argued that 802.1x security is flawed since it does not provide per-packet integrity and authenticity. Depending on setting, that may allow session hijacking, basically allowing an attacker to take over a MAC address that belongs to a legitimate and authenticated user.

We completely agree with the basic reasoning. To be properly secure in a shared medium environment, such as 802.11 WLAN, 802.1x authentication should be tightly integrated with a link-level integrity system that would use different session keys for different clients. However, today there are no standards for such link-level encryption. Besides, in some of the given scenarios, such as the pre-paid pay-per-use, the benefits a session hijacker might get from its attack may be questioned. However, the potential benefit depends on the specific characteristics of the underlying link-layer; if the link-layer makes it impossible for two clients to use the same MAC address at the same time, the attacker would have difficulties to use more than one pre-paid time slot.

If we really want to secure the wireless link against intruders, there are a few possible approaches. One would be to develop further the proposed IEEE Robust Security Network (RSN) architecture along the lines suggested in [15]. That would involve using the Wireline Equivalent Privacy (WEP) enhanced with changing keys derived from keying material received via 802.1x. Another possibility would be to use IPsec Authentication Header (AH) between the client and the access point, deriving the AH session keys from the keying material provided by e.g. TLS over EAP. Still a different approach would be to use IPsec IKE instead of 802.1x in authenticating the clients, and then use AH. However, all such approaches are beyond the scope of this paper.

Thus, within the scope of this work we have limited ourselves to checking the MAC addresses. We acknowledge that this is not always sufficient and definitely not the right long term solution. However, for our main scenario, pre-paid pay-per-use, the achieved security level seems appropriate for the time being. The only additional security feature that we are considering to add in the future is to combine MAC level and IP level filtering. That blocks simpler attacks where the attacker just sends packets using the same MAC address as some other node but a different IP address.

7.2 From OTP to micropayments

From the open source and grassroots movement point of view, the most lucrative usage scenario would be one where the good properties of the closed community scenario and the pre-paid pay-per-use scenario would be combined. However, it is not at all clear how such a scheme could be achieved. For example, we have considered to putting a full-fledged three-party micropayment protocol over the top of EAP, and passing micropayment tokens from the client to the access point. That would allow the client to pay to the access point, and the access point owner would then be able to use the tokens collected by the access point somewhere else. Unfortunately there seems to be problems in the trust structure, and more research is needed to figure out how such a structure and decentralized administration could be combined.

8 Conclusions

In this paper, we have shown that the IEEE 802.1x standard can be used to solve useful problems beyond its original purpose. Furthermore, we have shown that the FreeBSD netgraph facility makes it very easy to implement the protocol, and at the same time to provide a flexible architecture that allows the same software modules to be applied to widely varying purposes. In particular, we have illustrated that, in addition to user authentication, unmodified 802.1x can be used for pre-paid, pay-per-use access. We have also shown that is fairly easy to support multiple operators at a single Wireless LAN using the FreeBSD netgraph facility.

Acknowledgments

I am indebted to David Partain and Per Magnusson of Ericsson Research, who worked for a few months with me at the iPoints innovation cell. Many of the central ideas for this paper were developed together with them during that time. Thus, whenever I am referring to collective effort in this paper, I am usually referring to the work conducted together with David and Per. However, I alone am responsible for any mistakes and shortcomings in this paper.

I also want to thank Juha Heinänen of Song Networks for pointing out the importance of 802.1x in the first place, and especially for his insights and ideas related to the multioperator scenario. I am also thankful to Julian Elischer for his help in implementing the netgraph modules, and to Craig Metz and Marco Moteni for their comments on various versions of the paper.

Availability

An alpha version of the EAPOL code is available at <http://www.tml.hut.fi/~pnr/eapol/>. The distribution includes the netgraph modules and simple EAP OTP authenticator and supplicant user level programs, together with some scripts used for testing and performance measurements.

References

- [1] *IEEE Draft P802.1X/D11: Standard for Port based Network Access Control*, LAN MAN Standards Committee of the IEEE Computer Society, March 27, 2001.
- [2] L. Blunk and J. Vollbrecht, *RFC2284, PPP Extensible Authentication Protocol (EAP)*, IETF, March 1998.
- [3] C. Rigney, S. Willens, A. Rubens, W. Simpson, *RFC2865, Remote Authentication Dial In User Service (RADIUS)*, IETF, June 2000.
- [4] Archie Cobbs, *All about netgraph*, daemon news, March 2000, <http://www.daemonnews.org/200003/netgraph.html>
- [5] Nick Petroni, Bryan D. Payne, Bill Arbaugh, and Arunesh Mishra, *Open1x project home page*, University of Maryland, February 2002, <http://www.open1x.org>,
- [6] B. Aboba, D. Simon, *RFC2716, PPP EAP TLS Authentication Protocol*, IETF, October 1999.
- [7] N. Haller, C. Metz, P. Nesser, M. Straw, *RFC2289, A One-Time Password System*, Internet Engineering Task Force, February 1998.
- [8] Seattle Wireless Home Page, <http://seattlewireless.net/>
- [9] Electrosnog Home Page, <http://elektrosmog.nu/>
- [10] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis, *RFC2704, The KeyNote Trust-Management System Version 2*, Internet Engineering Task Force, September 1999.
- [11] OpenSSL Home page, <http://www.openssl.org>
- [12] NoCatAuth White Paper, <http://nocat.net/nocatrfc.txt>
- [13] Rob Flickenger, *NoCatAuth: Authentication for Wireless Networks*, O'Reilly Network Wireless Devcenter, November 9th 2001, <http://www.oreillynet.com/pub/a/wireless/2001/11/09/nocat-auth.html>
- [14] John Saunders, *transproxy*, <ftp://ftp.nlc.net.au/pub/unix/transproxy/>
- [15] Arunesh Mishra and William A. Arbaugh, "An Initial Security Analysis of the IEEE 802.1X Standard", UMIACS-TR-2002-10, University of Maryland, February 2002.

ACPI implementation on FreeBSD

Takanori Watanabe
Kobe University

Abstract

Prior to the introduction of the Advanced Configuration and Power Management Interface (ACPI), PCs did not have a unified standard mechanism that allowed the operating system to enumerate, configure, and manage the power usage and thermal properties of built-in hardware devices. Instead, these devices were either left unmanaged, or they were managed by special BIOS-level code such as Plug-and-Play BIOS (PnP BIOS), Advanced Power Management BIOS (APM), or other vendor-specific BIOS code. These firmware-driven methods increase firmware costs, and the resulting BIOS code is difficult to alter or debug. Device management issues are becoming more important, especially in mobile computing environments where fine-grain power management is often necessary. ACPI replaces PnP BIOS, APM, and a number of ad hoc methods while providing a management framework that allows increased flexibility in hardware design. Unfortunately, the increased power and flexibility of ACPI comes with a cost: it requires substantial software support from the operating system kernel. In this paper we describe ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI.

1 Introduction

Almost all modern computer system hardware allows its power usage to be managed and its temperature to be monitored and kept at the appropriate level. This allows users to achieve the best performance from a system for a given level of power usage. This is especially important for battery-driven mobile platforms where unnecessary use of power reduces battery life and thus reduces the amount of work that can be done before a recharge is necessary. It is also useful in desktop environments where devices such as computer monitors and disk drives can be powered down during idle times to reduce energy consumption.

Unfortunately, in the typical PC most of the software that configures and manages the power and thermal environment within the computer is locked up within the BIOS. In addition, unless a power management device is attached to a PnP bus (e.g. PCI), the operating system has no easy way to detect, configure, or manage it. For example, mechanisms such as ISA PnP are usually used to enumerate add-on ISA cards rather than for on-board devices. Systems like PnP BIOS can be used to enumerate on-board devices, but it is hard to extend in a generic way. Also, PnP BIOS is written in 16-bit code, so the operating system must use 16-bit emulation in order to call PnP BIOS functions.

Prior to the introduction of the Advanced Configuration and Power Management Interface (ACPI), the Advanced Power Management (APM) BIOS was commonly used for power management. In APM, the bulk of the power management control and logic resides within the APM BIOS code itself. APM-aware operating systems communicate with the APM BIOS through a fixed BIOS API which provides basic access to BIOS functions. APM-aware operating systems must periodically poll APM for APM-related events that must be processed. The APM BIOS may also make use of special system management interrupts which are invisible to the operating system itself. APM provides four states: run, suspend, sleep and soft-off.

APM has three main limitations. First, without special vendor programs, many APM features are only available through vendor-specific BIOS menus before the operating system is loaded. For example, the amount of console idle time required before powering down the video display is usually configured this way. Also, with APM, the number of power management configurations are fixed by the BIOS vendor. For example, the APM BIOS may always slow the CPU clock or power down other devices (e.g. networking card) when powering down the monitor. Since this is under control of the BIOS, there is no way to change the policy without changing the BIOS.

Second, APM is BIOS-level code that operates outside of the scope of the operating system. This makes developing and debugging APM code a challenge. It also means

that users can only fix bugs in their APM BIOS by flashing a new one into ROM. Flashing a new BIOS is a dangerous operation because if the BIOS fails, the system may well become useless.

Third, as APM is vendor-specific, efforts to develop and maintain the complex APM BIOS code are duplicated across the vendors that use it. This is wasteful.

ACPI addresses the limitations of APM and other configuration mechanisms by unifying all device management within the operating system kernel rather than having BIOS code make most of the decisions. Thus, ACPI is said to allow “operating system directed” power and thermal management that is more flexible than other mechanisms. The cost of this flexibility is the extra complexity required in the kernel to support ACPI. In this paper we describe ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI. In Section 2 we describe the general architecture of ACPI. Section 3 explains the architecture of the FreeBSD ACPI implementation and its impact on the rest of the kernel. Section 4 contains related work, and Section 5 has our conclusions and future work.

2 ACPI Architecture

The ACPI standard [1] was developed by Intel, Toshiba, and Microsoft and has been adopted by most PC manufacturers. Figure 1 shows the main components of a system that uses ACPI. At the lower level, the ACPI standard defines a set of tables that describe the hardware platform, a BIOS API for low-level management operations, and a pre-defined set of registers. In the upper level, the operating system contains some core software and drivers used to communicate with ACPI in addition to the usual device framework and drivers used to manage non-ACPI devices. In this section we examine both levels of the ACPI architecture.

2.1 ACPI Specified Components

The ACPI Specification defines three types of components:

ACPI tables: The ACPI tables are the central data structure of an ACPI-based system. They contain definition blocks that describe all the hardware that can be managed through ACPI. These definition

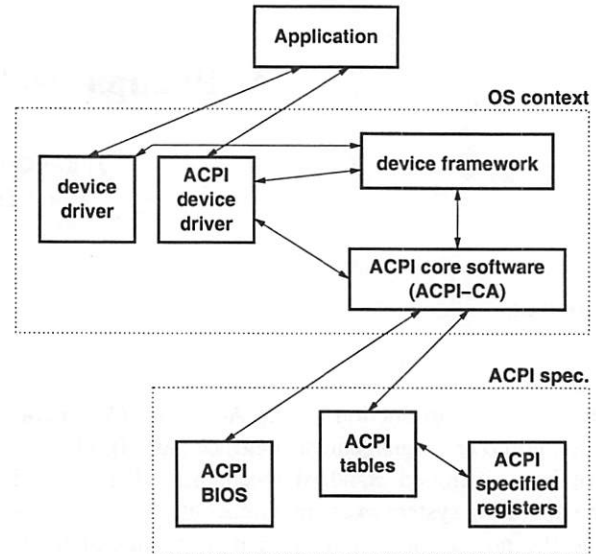


Figure 1: ACPI Architecture

blocks include both data and machine-independent byte-code that is used to perform hardware management operations.

ACPI BIOS: The ACPI BIOS is a small BIOS that performs basic low-level management operations on the hardware. These operations include code to help boot the system and to put the system to sleep or wake it up. Note that the ACPI BIOS is much smaller than an APM BIOS because most of the management functions have moved into the operating system and ACPI tables.

ACPI registers: The ACPI registers are a set of hardware management registers defined by the ACPI specification. The address of these registers is located through definition blocks in the ACPI tables. Note that hardware designers may provide additional management registers beyond the ones defined in the ACPI specification. These additional registers can be located and accessed through the byte-code stored in the device-specific part of the ACPI tables.

When an ACPI-based system is powered up, before the operating system is loaded the ACPI BIOS places the initial ACPI tables in memory. Since the ACPI tables are typically too large to put in the 128KB BIOS memory area, the ACPI BIOS obtains a physical memory map of the system in order to allocate space for the ACPI tables. When an ACPI-aware operating system kernel is started, it search for a small data structure within the BIOS memory area. If a valid structure is found (e.g. if

its checksum and signature match) then the kernel uses this structure to obtain a pointer to the ACPI tables and memory map. This information is used by the kernel to preserve the ACPI tables when the virtual memory system is started.

The definition blocks within the ACPI tables are stored in a hierarchical tree-based name space. Each node in the tree is named. Node names consist of four capital alphanumeric characters and underscores (e.g. “FOO_,” or “_CRS”). Namespace components are separated by periods, and the root of the namespace is denoted with a backslash (“\”). Names without a leading backslash are considered to be relative to the current scope in the name space. Node names that begin with an underscore are reserved by the ACPI specification for describing features. For example, nodes in the _SB namespace refer to busses and devices attached to the main system bus, nodes in the _TZ namespace relate to thermal management, and nodes in _GPE are associated with general purpose ACPI events.

Except for the few operations performed by the ACPI BIOS, almost all ACPI operations are performed in the operating system context by interpreting machine-independent ACPI Machine Language (AML) byte-code stored in the ACPI tables. These blocks of AML are called methods. AML methods are stored in specially named nodes in the ACPI namespace. For example, the name _PS0 is reserved for storing AML methods that evaluate a device’s power requirements in the “D0” state (device fully on). Thus the node _SB.PCI0.CRD0._PS0 contains an AML _PS0 method for the CRD0 device on the system’s PCI0 bus.

AML is usually compiled from human-readable ACPI Source Language (ASL). Figure 2 shows an example block of ASL code for thermal management that defines four named data elements and two methods. The “Scope” operator defines what part of the ACPI namespace the contained block of code resides in. The “ThermalZone” operator defines a object representing a region of thermal control. The “Device” operator defines a device object, and the “PowerSource” operator defines a power switch object. The “OperationRegion” and “Field” operators are used to define blocks of registers and fields within them, respectively. The “Name” and “Method” operators define data and program elements belonging to their parent objects. For example, the first “Name” in the figure defines _TZ.TMZN._AC0 (the fan high-speed threshold) to be the integer 3272, which means 327.2 K. The “_ON” method defined in the figure contains code to turn the fan on. A graphical representation of the namespace defined in Figure 2 is shown in

```
Scope(\_TZ){
    ThermalZone(TMZN){
        Name(_AC0, 3272)
        Name(_AL0, Package{FAN})
        ....
    }
    Device(FAN){
        Name(_HID, 0xb00cd041)
        Name(_PR0, Package{PFAN})
    }
    OperationRegion(FANR, SystemIO,
                    0x8000, 0x10)
    Field(FANR, ByteAcc, NoLock,
          Preserve){
        FCTL, 8
    }
    PowerSource(PFAN, 0, 0){
        Method(_ON){
            Store(0x4, FCTL)
        }
        Method(_OFF){
            Store(0x0, FCTL)
        }
    }
}
```

Figure 2: Example ASL Code

Figure 3.

In the next three subsections we describe ACPI’s configuration, power management, and thermal management subsystems.

2.1.1 Configuration

The ACPI namespace contains a tree of devices attached to the system. ACPI-aware operating systems walk this tree to enumerate devices and gain access to the device’s data and control methods. In the ACPI namespace, an ACPI device description consists of a device object node and its children. Common children of device nodes include:

- _ADR:** a bus-specific address. For example, this could be a PCI device and function number.
- _HID:** the EISA ID of an on-board device. This is used to identify the device.
- _UID:** the unit number of an on-board device. This is used to distinguish between same kind of device.

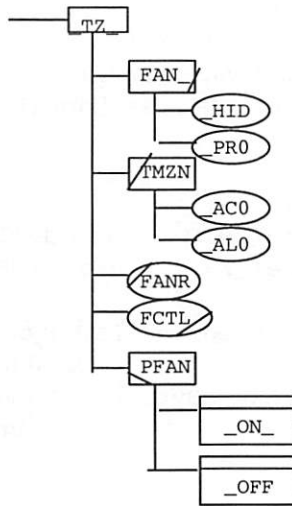


Figure 3: Example ACPI Namespace

_CRS: the current resource settings. This method produces a byte stream that is similar to a PnP resource encoding.

_PRS: the possible resource settings. This method also produces a byte stream similar to a PnP resource encoding.

_SRS: set resource settings. This method takes a PnP-encoded byte stream and uses it to set the device's configuration.

Note that the operating system accesses device data and methods through the ACPI-CA software layer (described in Section 2.2).

2.1.2 ACPI Power Management

Hardware power management events trigger an OS-visible interrupt called a “system control interrupt” (SCI). Operating systems handle simple SCI interrupts (e.g. fixed-feature power button state change) directly. Complex SCI interrupts are handled by the OS using AML code associated with the interrupt. For example, consider what happens when a “sleep” SCI interrupt occurs. The kernel must first save hardware state. The kernel then calls the `_PTS` (prepare to sleep) method. Finally, it puts the system to sleep by writing the appropriate value to an ACPI register.

In ACPI there are six power states: S0, S1, S2, S3, S4, and S5. These states are defined as follows:

S0: the run state. In this state, the machine is fully running.

S1: the suspend state. In this state, the CPU will suspend activity but retain its contexts.

S2 and S3: sleep states. In these states, memory contexts are held but CPU contexts are lost. The differences between S2 and S3 are in CPU re-initialization done by firmware and device re-initialization.

S4: a sleep state in which contexts are saved to disk. The context will be restored upon the return to S0. This is identical to soft-off for hardware. This state can be implemented by either OS or firmware.

S5: the soft-off state. All activity will stop and all contexts are lost.

In addition to managing transitions between system power states, ACPI can also manage the power state of individual devices to a fine-grained level. For example, if two devices share the same power line, that information can be encoded in the ACPI tables in such a way that the power line is active only if one or both of the devices are in use.

2.1.3 Thermal Management

ACPI supports operating system directed thermal management. Prior to ACPI there was no unified interface for thermal management (e.g. APM did not support it). On some systems the platform specific BIOS code handles thermal management, but this is invisible to the operating system and few OS developers noticed it. Thermal management is becoming more important as system efficiency increases and the system gets hotter when working.

The ACPI thermal management subsystem provides a way to get the current temperature, and it provides hints to control thermal policy. The thermal policy information includes information on how to get the system cooler and at what point the cooling method should be invoked. There are two ways to cool the system down: active cooling, which activates cooling devices such as fans, and passive cooling, in which the CPU operation slows down to decrease heat generation.

Thermal management is controlled in the ThermalZone section of the AML namespace. Note that in ACPI, all temperatures are in tenths of degrees kelvin. Important

sections of the thermal management part of the ACPI namespace include:

- _TMP:** gets the current temperature.
- _ACx:** the temperature at which the system should switch to active cooling mode “x.”
- _ALx:** a list of objects that should be active when the system is in cooling mode “x.”
- _CRT:** the temperature at which we should halt the entire system.
- _PSV:** the temperature at which the system should switch to passive cooling mode.
- _PSL:** a list of objects (typically the CPU) that should be slowed in passive cooling mode.
- _SCP:** a method that allows the operating system to set the cooling policy.
- _TCx:** a parameter for passive cooling mode “x.”

Note that in the event of a thermal emergency (e.g. a bug in ACPI software), ACPI allows the hardware to take over thermal management in order to protect the hardware from damage.

2.2 ACPI Component Architecture (ACPI-CA)

An ACPI-aware operating system must include code that accesses the ACPI BIOS, registers, and tables. It must also include an AML byte-code interpreter. This upper layer of core ACPI software is shown in Figure 1. Intel has implemented an OS-independent implementation of this layer of software called ACPI Component Architecture or ACPI-CA [2]. ACPI-CA is used by many open source operating systems including FreeBSD and Linux.

ACPI-CA provides a high-level ACPI API to the operating system. The OS uses this API to implement power management, device configuration, and thermal management. All fixed features and some access to AML names are wrapped by the exported function. But some ACPI-specific devices have to access ACPI namespace. The ACPI-CA API is shown in Table 1.

Operating systems that use ACPI-CA must provide it with some basic low-level functions. Intel has implemented the low-level part for Linux. A list of these functions is shown in Table 2.

One of the main user-visible differences between FreeBSD ACPI and Linux ACPI is the user interface. FreeBSD uses `sysctl` to export ACPI-related kernel variables, while Linux uses the `procfs` /`proc` filesystem to export them.

3 Design of FreeBSD ACPI

In this section we describe how we made the FreeBSD kernel ACPI-aware. We first implemented our own version of the ACPI core software (we later switched to ACPI-CA). We then addressed the issues of ACPI device enumeration, supporting ACPI sleep modes, and ACPI thermal management.

3.1 Our ACPI Core Software Implementation

In September 1999 we started writing our own ACPI core software implementation, including an AML execution environment. The implementation was based on Doug Rabson’s ACPI disassembler and our ACPI data analyzing tool.

We first wrote a ACPI memory recognition routine to detect and preserve the ACPI tables. We then wrote a process that could run AML methods manually (e.g. suspend and wakeup) based on somewhat incomplete ASL output. This allowed us to enter power state S1 and also to shutdown a machine by pushing the power button.

We also wrote an AML interpreter in user space by merging the namespace functions from our analyzing tool into the ACPI disassembler and adding a memory management module to it. After this was implemented we merged the AML interpreter module into a kernel driver and then we had a basic working version of power management.

While we were working out the bugs in our in-kernel AML interpreter, we noticed that ACPI-CA software from Intel had a suitable license to merge into FreeBSD. As we were preparing to merge our ACPI into the main branch of the FreeBSD source repository we read the ACPI-CA implementation. We then decided to switch to ACPI-CA using glue code that we wrote. The reason we switched was that ACPI-CA is an OS-independent implementation so we can share and benefit from feedback from other groups. While the ACPI-CA implementation is larger, it is also more complete and well documented.

Function Class	Functions
ACPI subsystem management	AcpiInitializeSubsystem, AcpiEnableSubsystem, AcpiTerminate, AcpiSubsystemStatus, AcpiDisable, AcpiGetSystemInfo, AcpiFormatException, AcpiPurgeCachedObjects
memory management	AcpiAllocate, AcpiFree
ACPI table management	AcpiFindRootPointer, AcpiLoadTables, AcpiLoadTable, AcpiUnloadTable, AcpiGetTableHeader, AcpiGetTable, AcpiGetFirmwareTable,
namespace interface	AcpiWalkNamespace, AcpiGetDevices, AcpiGetName, AcpiGetHandle, AcpiAttachData, AcpiDetachData, AcpiGetData
object manipulation	AcpiEvaluateObject, AcpiGetObjectInfo, AcpiGetNextObject, AcpiGetType, AcpiGetParent
event handler interface	AcpiInstallFixedEventHandler, AcpiRemoveFixedEventHandler, AcpiInstallNotifyHandler, AcpiRemoveNotifyHandler, AcpiInstallAddressSpaceHandler, AcpiRemoveAddressSpaceHandler, AcpiInstallGpeHandler, AcpiAcquireGlobalLock, AcpiReleaseGlobalLock, AcpiRemoveGpeHandler, AcpiEnableEvent, AcpiDisableEvent, AcpiClearEvent, AcpiGetEventStatus,
resource interfaces	AcpiGetCurrentResources, AcpiGetPossibleResources, AcpiSetCurrentResources, AcpiGetIrqRoutingTable,
hardware interface	AcpiSetFirmwareWakingVector, AcpiGetFirmwareWakingVector, AcpiEnterSleepStatePrep, AcpiEnterSleepState, AcpiLeaveSleepState

Table 1: ACPI-CA API

So our implementation is no longer in the kernel, but it still remains in user-level tools such as `aml.db(8)` and `acpidump(8)`.

3.2 FreeBSD ACPI Device Enumeration

FreeBSD uses a configuration system known as “new-bus.” [5] In this system, an opaque “device_t” type object represents each bus/device. There are functions to manipulate device_t objects. These functions (e.g. probe, attach, allocate resources) are device specific. They are invoked through a function table that acts as an associative array. The array key is called a method. This is the same technique used in the BSD virtual filesystem layer (VFS). Each device has a parent device, except for the root device. Each device_t object has two device-specific structures: “ivar” and “softc.”

The softc structure is a device-specific structure used to store a device’s state. Each driver determines the size of its own softc structure, and the new-bus framework allocates memory for the softc structures as devices are configured.

The “ivar” structure is used by a parent device to manage its children. This variable should not be accessed from the child device directly, but via the parent device method. The child device uses this method to obtain

bus-specific values and resources. An example of this method is `BUS_READ—WRITE_IVAR`, which is used to access a bus-specific value. This function is usually wrapped by macro-definition in a bus-specific header file.

PnP devices are processed by a “pnp” or “pnpbios” driver that provides only the `DEVICE_IDENTIFY` method. The `DEVICE_IDENTIFY` method is usually called from a parent bus’s routine, after the bus was probed and before the actual attach process, via the `bus_generic_probe` function. This method records the device’s logical id in the ISA bus-specific ivars structure. This value is then used when the `isa_get_logicalid` macro is invoked. This macro calls the `BUS_READ_IVAR` bus method to get the ID.

In FreeBSD ACPI, all device objects, thermal zone objects, and some other fixed features are added as child devices in the acpi bus attach code. This is done with the “device_add_child” function. Once an ACPI child device is added, this function sets ACPI object handles in the ivars. The probe routine check to see whether the `_HID` of the device will match the driver. If it does, then the attach routine calls the resource parser to get the resources the driver will use.

The Host-PCI bus bridge appears in the ACPI namespace and is treated as an ACPI-specific device. The driver will call the machine-dependent PCI bridge func-

Function Class	Functions
library initialization	AcpiOsInitialize, AcpiOsTerminate
semaphore control	AcpiOsCreateSemaphore, AcpiOsDeleteSemaphore, AcpiOsWaitSemaphore, AcpiOsSignalSemaphore
memory allocation	AcpiOsAllocate, AcpiOsCallocate, AcpiOsFree, AcpiOsMapMemory, AcpiOsUnmapMemory, AcpiOsGetPhysicalAddress
interrupt control	AcpiOsInstallInterruptHandler, AcpiOsRemoveInterruptHandler
process control	AcpiOsGetThreadId, AcpiOsQueueForExecution, AcpiOsSleep, AcpiOsStall
device access	AcpiOsReadPort, AcpiOsWritePort, AcpiOsReadMemory, AcpiOsWriteMemory, AcpiOsReadPciConfiguration, AcpiOsWritePciConfiguration, AcpiOsReadable, AcpiOsWritable
signal/timer control	AcpiOsGetTimer, AcpiOsSignal
diagnostic functions	AcpiOsPrintf, AcpiOsVprintf, AcpiOsGetLine, AcpiOsDbgAssert

Table 2: OS functions provided to ACPI-CA

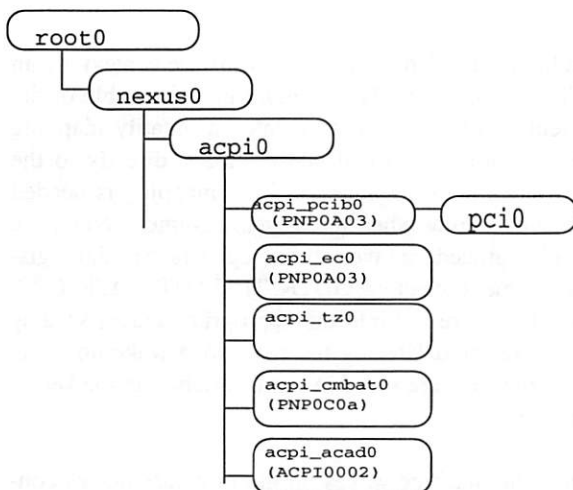


Figure 4: FreeBSD device tree corresponding to the ACPI name space

tion to implement a bus-bridge function and manage PCI interrupt routing with ACPI. In ACPI, PCI routing information is in a `_PRT` object. A `_PRT` object contains an array of structured objects, each of which has information about the slot, pin, and interrupt source. The interrupt source points to the device object of the interrupt router and assigns the interrupt resource to the device route interrupt to the pin. The interrupt router device object is not currently treated as a FreeBSD new-bus device. The ACPI device driver mimics some ISA-like behaviors. This responds to the `isa_get_logicalid` function and answers the device logical id value, using `_HID` object. Figure 4 shows device tree constructed from ACPI name space.

Prior to September 2001 a different approach was taken.

The earlier approach was closer to a PnP driver. The “acpi.isa” driver was used for this. Like a PnP driver, the “acpi.isa” driver provides only the `DEVICE_IDENTIFY` method.

3.3 FreeBSD ACPI Power States

In this section we describe how FreeBSD implements ACPI power states. We start with some background of the i386 architecture before going into the details of the FreeBSD implementation.

3.3.1 I386 Background

To understand ACPI power states, it is important to be familiar with i386 CPU architecture and CPU initialization. All i386 CPUs support “real mode.” Real mode is a CPU state that is compatible with the old i8086 processor. In this mode each memory access pointer, including the instruction pointer, is 16 bits long. This provides a 64KB address space. Segment registers are used to extend the memory address space that can be referenced and also to separate the code, data, and stack areas. To convert a segment-based address into a physical address, take the segment register value and shift it left four bits and add it to the address. The extra four bits from the segment register allow us to reference up to 1MB of physical memory. There are six segment registers: CS, DS, SS, ES, FS and GS. The code segment register (CS) is used when accessing instructions through the instruction pointer. When accessing data memory, the data segment register (DS) is used by default. Stack operations use the stack segment register (SS).

When booting, the BIOS firmware passes program control to software in real mode, as described above. Most modern operating systems do not operate in real mode. Instead, they change the working mode from real mode to “protected mode.” To switch to the protected mode, the kernel must set up the GDTR register and update control register CR0. The GDTR register points to the global descriptor table (GDT). The GDT is used for address translation in protected mode.

In protected mode, segment registers and some special registers, such as TR (Task register), point to an entry in the GDT. This entry is then used when translating addresses. The current mode of the CPU is determined by the mode-select flag in the CR0 register. Once the system is in protected mode, kernels that wish to use paging for virtual memory must enable it. To enable paging, the kernel has to set the CR3 register to point to the page table structures to use and then change the mode flag in the CR0 register.

3.3.2 FreeBSD ACPI Sleeping States

The S1 state is implemented simply by calling an ACPI-CA function after sending the device-sleeping request to the device driver. This ACPI-CA function uses the ACPI registers to stop the system. When the system wakes up from S1 sleep, it can immediately resume processing from where it was just before the sleep request was made.

The S2 and S3 states do not preserve the CPU context. So the FreeBSD kernel is required to do a CPU context save is required before entering the sleep state.

Since entering and exiting the S3 sleep state requires the use of real-mode, the kernel must place a “resume handler” somewhere in the first 1MB of memory so that it can be addressed from real mode. We use a perl script along with objcopy, hexdump, and nm to generate code. With this script, a real-mode executable object is turned into a header file with an array of chars and some definitions that point to the offset of the symbol in the object file. When the system boots the memory for the resume handler is allocated in the first 1MB as early as possible using a special memory allocator. The ACPI driver attach routine then copies and links in the resume handler code to the newly allocated memory area. Finally, the physical address of the resume handler is recorded in the ACPI driver software context.

When S3 is invoked, the physical address of the resume

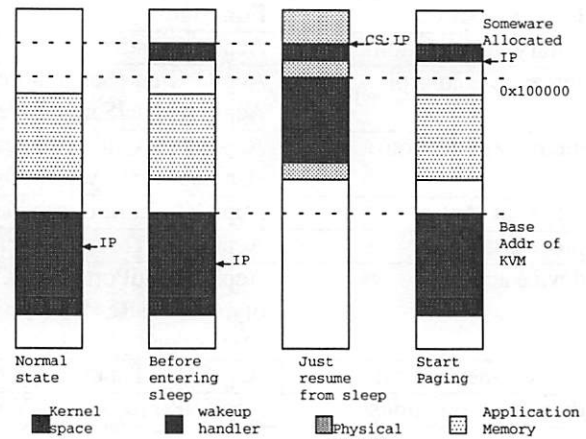


Figure 5: Memory mapping transition

handler is installed in the ACPI firmware context via an ACPI-CA function. The memory mapping table of the current process is used to create an identity mapping where a process’ virtual address maps directly to the hardware’s physical address. This mapping is needed to restart paging when the system resumes. Next, the sleeping procedure saves all the registers. Special registers (segment selectors, GDTR, TR, LDTR, IDTR, CR2, CR3, CR4) are put into the appropriate places so they can be restored later by the real mode wake-up code. Other registers are saved in static variables in the kernel data area.

When the machine wakes up the firmware passes control, in real mode, to the resume handler. This is done by placing the top 16 bits of the physical address of the resume handler in the code segment register and placing the lower 4 bits in the instruction pointer. The handler sets the other segment registers to the same value as code segment registers. The CPU state can then be switched to protected mode by changing CR0. At this point the special registers are restored. Note that before restoring the task register, the task selector entry in GDT should be fixed up so that it does not become marked as a busy task selector.

After the special registers have been restored, then paging can be reenabled by setting the appropriate bit in CR0. After the paging has started, the instruction pointer is still pointing to an identity-mapped page. Next, the handler passes the control back to its calling routine which resides in normal kernel virtual memory. Finally, the stack pointer and remaining preserved registers are restored. Figure 5 shows memory mapping transition.

After returning to the sleep code, hardware devices

should be restored. First, the interrupt controller must be reprogrammed because some device drivers and ACPI itself depend on functioning interrupts. To do this, we split the `isa_defaultirq()@intr_machdep.c` function into two. Now `isa_defaultirq()` installs a stray interrupt handler, and then calls `init_i8259` to initialize the interrupt controller. We call the resumption function `icu_reinit()@intr_machdep`, and then we call the routine `init_i8259()`. We then check the interrupt handler and enable the interrupt if the handler is not a stray interrupt handler. After this call, we use the routine call `DEVICE_RESUME` method of `root.bus` to request that the device drivers resume the device, after which normal operation resumes.

3.4 FreeBSD ACPI Thermal Management

The thermal zone is represented as a device in FreeBSD. The FreeBSD thermal zone device driver exports thermal information using the `sysctl` interface. Passive cooling is not yet implemented, though CPU throttling is implemented. The device driver checks the thermal zone when a tunable polling time expires and also when thermal zone Notify AML opcode are evaluated. Notify opcodes are typically found in the general purpose interrupt handlers or in the query handler for ACPI-capable embedded controllers. ACPI routines check the thermal zone by fetching the temperature using the thermal zone `_TMP` method and comparing its value to the `_ACx` value. If the `_ACx` value is smaller than current temperature, then we change thermal mode to the largest `x` value. If the new thermal state value is larger than old one, we activate the device object listed in the `_ALx` object where `x` is the new state number, otherwise we deactivate them.

4 Related Work

In this section we compare FreeBSD ACPI to Linux ACPI and we examine the relationship between Open Firmware and ACPI.

4.1 Comparison with Linux ACPI

As both FreeBSD and Linux use Intel ACPI-CA, the basic architecture of their ACPI subsystems is similar. The major differences are in user interface and bus enumeration. For user interface, FreeBSD uses `sysctls`, which are

variables that kernel exports. Linux uses the “`procfs`” filesystem. While both system have `procfs`, FreeBSD uses `procfs` purely for query process information. The `sysctl` interface is interface provides a tree of for kernel tunable variable. We export some ACPI information such as temperature to userspace via the `sysctl` interface.

The ACPI-CA code is currently distributed with a user-space ACPI interpreter which is the counterpart of FreeBSD’s `amlldb(8)` and ASL assembler. But ACPI-CA has no disassembler tool (like FreeBSD’s `acpidump(8)`) that produces ASL compatible with the ACPI-CA ASL assembler.

In Linux, there were no generalized ways to create device trees, so Intel used a “bus manager” mechanism to recognize ACPI-specific devices. The bus manager abstraction was introduced when the ACPI-CA was developed and build under WIN32. We decided not to use it early in our development process because it collides with the FreeBSD driver recognition mechanism. But the Intel Linux-ACPI team recognized the necessity of the unified mechanism to configure devices (including non-ACPI ones), so they proposed a mechanism called “Linux Driver Model.” This mechanism will be introduced in next major version of the Linux kernel (2.5).

FreeBSD uses a three stage boot loader. Currently the FreeBSD boot loader detects ACPI by scanning the BIOS memory and loads the `acpi` kernel module automatically if it is needed. Linux use `initrd` mechanism to do early configuration. `Initrd` is special memory filesystem loaded by the boot loader. This file system is mounted as root for initial configuration such as module loading. After the configuration is finished, the file system is unmounted or moved to another mount point.

4.2 Comparison with Open Firmware

Open Firmware was originally developed by Sun Microsystems and is now standardized as IEEE 1275-1994 [6]. It is currently used in the SPARC architecture, the PowerPC architecture, and the ARM architecture. Open Firmware acts as monitor that can interpret the Forth language. It covers the boot process, device configuration, and power management. Operation systems running on Open Firmware based machines must implement some architecture-dependent way to access the firmware interpreter for use in auto-configuration.

ACPI and Open Firmware are similar in that some func-

tions of the firmware are written for an interpreter. This makes it easier to extend without breaking compatibility. But the most important difference is that ACPI byte code is interpreted by the operating system, while Open Firmware Forth code is interpreted by the firmware itself. The Open Firmware solution provides a powerful framework to describe and extend functionally, but this is not accepted in Intel architecture (IA32/IA64) for the following reasons:

- Firmware space is limited for compatibility reasons.
- Firmware cannot figure out all states in the devices, which is especially needed to implement suspend state.
- Calling firmware from 32 bit code is somewhat unstable in Intel architecture. There is no compatible way to setup without calling 16bit code. And there is no way to notify event other than polling.

5 Conclusion and Future Work

In this paper we have described ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI. We believe that ACPI support will become more important as new devices with demanding configuration, power, and thermal management needs become more widespread. Although we have a basic working ACPI environment under FreeBSD there is still lots of work left to do.

5.1 Device Enumeration Enhancement

ACPI has a hot-plugging feature, and the current PCI interrupt routing code is not capable of routing interrupts for devices that are on a PCI-PCI bridge. This will require large modification to the current device enumeration scheme in ACPI. We have proposed a scheme designed so that existing drivers are not modified unless absolutely necessary. The scheme is:

1. Add a bus bridge enumerator driver (only have device.identify bus method) for each bus bridge device that can be a descendant of ACPI and appear as a namespace. In this method, add children to the bus and register `acpi_name-device.t` table in the

acpi driver. Then evaluate `_INI` object after checking by `_STA`. Then the driver install address space handler, etc.

2. The manipulation to get `ACPI_HANDLE`, etc., is not done via `DEVMETHOD` but by a direct function call. This may require module dependency with the acpi driver, but if the driver wants to use `ACPI_HANDLE`, it must depend on acpi. If a device other than ACPI is used, it may not use `ACPI_HANDLE` to get information.
3. Add acpi attachment for devices that can be attached to acpi directly. The acpi-pci driver is quite a different implement than the nexus-pci driver.

5.2 User-land Interface

There are already some user-land interfaces in the ACPI driver. The interface is provided in two ways: `/dev/acpi` ioctl's and `hw.acpi` sysctls. Currently provided interfaces are:

- Battery charge information.
- Thermal zone information.
- CPU speed configuration.

There is a piece missing relating to the ACPI event notification system. We plan to use `kqueue` [7] to implement the event notification system.

5.3 S4 Implementation

The S4 sleep state is also known as "Suspend to Disk." ACPI S4 implementation is achieved in two ways. The first way is called S4 BIOS: Firmware saves the running state to disk, and the operating system should do the same thing as S2/S3, then issue a suspend request to BIOS via the special port that triggers a system management interrupt. The second way to handle S4 is to have the operating system handle the saving of state to disk. In this scheme, the operating system should preserve all memory contents, device contexts, and the CPU context. FreeBSD includes a crash dump mechanism that we believe can be adapted for use when implementing OS-initiated S4 sleep.

Acknowledgments

IWASAKI Mitsuru provided great help not only in the actual implementation but also in project management. Without his help, this project would have remained only a personal project and been forgotten without prominent achievement.

Michael Smith contributed to our experimental implementation and did great work on ACPI-CA migration. Most of the FreeBSD dependent parts of the ACPI-CA code are his work.

Andrew Grover and his Intel associates helped in the porting work. Of course, we greatly respect their work on ACPI-CA itself and appreciate their kindly allowing us to use their work.

Doug Rabson wrote the ACPI disassembler, on which our experimental ACPI implementation was based.

Yasuo Yokoyama, Munehiro Matsuda and all acpi-jp mailing list people have contributed to our experimental implementation.

The ACPI4Linux mailing list gave me useful information about ACPI and ACPI-CA.

Warner Losh and Chuck Silvers gave suggestions to improve the expression of English in this manuscript.

Chuck Cranor shepherded me for the USENIX Freenix track.

- [5] Murray Stosky et al., "FreeBSD Developer's Handbook," <http://www.freebsd.org/>
- [6] Sun Microsystems, "IEEE Std 1275-1994. Open Firmware Core Specification." <http://playground.sun.com/1275/>
- [7] Jonathan Lemon, "Kqueue-A Generic and Scalable Event Notification Facility," FREENIX Proceedings, 2001.

References

- [1] Advanced Configuration and Power Management Interface, <http://acpi.info/>
- [2] ACPI Component Architecture, Intel Corp., <http://developer.intel.com/technology/iapc/acpi/>
- [3] Takanori Watanabe, "ACPI in general and implementation of its driver," Personal Unix / FreeBSD Press, <http://www.ux.mycom.co.jp/> (in Japanese)
- [4] Intel Corp., "Intel Architecture Software Developers Manual," <http://developer.intel.com/>

Gscope: A Visualization Tool for Time-Sensitive Software

Ashvin Goel, Jonathan Walpole

Department of Computer Science and Engineering
Oregon Graduate Institute, Portland
{ashvin,walpole}@cse.ogi.edu

Abstract

This paper describes *gscope*, a visualization tool for time-sensitive applications. *Gscope* provides an oscilloscope-like interface that can be integrated with applications. It focuses on software visualization and is thus designed to handle various types of signal waveforms, periodic or event-driven, in single or multi-threaded environments as well as local or distributed applications. *Gscope* helps in visually verifying system correctness and modifying system parameters and thus can complement standard debugging techniques and be used to build compelling software demos. Initial experiments with using *gscope* show that the library has low overhead.

1 Introduction

Modern processor speeds and high speed networks have made multimedia and other timing-sensitive applications common on desktop computers. For instance, today a standard desktop computer comes equipped with a DVD player, DVD and CD burner, TV tuner and digital video editing and conferencing software, making it a full featured home audio and video client or server. Although these types of timing-sensitive applications are becoming common, implementing them is non-trivial because existing tools for visualizing and debugging alter the timing behavior. For instance, a standard debugger stops an application and thus affects its timing behavior.

Current techniques for visualizing, testing and debugging time-sensitive applications involve some or all of

these steps: 1) create an experimental setup, 2) generate data in real-time, 3) collect data and store it to files, 4) process the file data offline, and 5) plot the data. The first three steps are complicated by the fact that the programmer attempts to minimize the impact of these steps on the application's timing behavior. The programmer must often repeat these steps several times before being satisfied with the results. In addition, for a distributed application, data files must be collected from multiple machines and transferred to a single machine where the data is correlated before it can be processed.¹ The problem with this approach is that the visualization and debugging cycle is long and error prone. It is error prone because the steps outlined above are often not an integral part of the application. Further, with this approach, it is not easy to demonstrate or experimentally validate system behavior in real-time.

Unlike the ad hoc tools used for visualizing time-sensitive software, there exists a time-tested visualization tool in the hardware community: the *oscilloscope*. The invention of the oscilloscope started a revolution that allowed "seeing" sound and other signals, experiencing data, and gaining insights far beyond equations and tables [11]. Today, an oscilloscope, together with a logic analyzer, is used for several purposes such as debugging, testing and experimenting with various types of hardware that often have tight timing requirements. We believe a similar approach can be applied effectively for visualizing time-sensitive software systems.

We have implemented a software visualization tool and library called *gscope* that borrows some of its ideas from an oscilloscope. The *gscope* design is motivated by the following goals:

- Simplify visualization of system behavior in real-time, especially the interactions among concur-

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

¹In some cases, it is almost impossible to correlate distributed data for analysis, but we will assume that distributed data can be correlated.

rent or competing software components, within or across machine boundaries.

- Simplify visualization of system behavior in real-time, especially the interactions among concurrent or competing software components, within or across machine boundaries.
- Simplify modification of system behavior in real-time.
- Enable building compelling software demos that can help explain the internal working of a time-sensitive system.
- Enable visual verification of system correctness.
- Complement standard debugging techniques with a real-time “debugging” tool.
- Build an easy to use library, thus encouraging use of visualization as an integral part of the application.
- Build a generic and extensible library that does not need specific hardware for correct operation.
- Build free software that is available to all users.

From an ease of use perspective, the oscilloscope interface is ideal. The probes of the oscilloscope are hooked to a circuit and, loosely speaking, the oscilloscope is ready for use. Our goal is to emulate this simplicity in interface as much as possible while extending it when needed to accommodate software needs. In the simplest case, a *gscope* signal consists of a signal name and a word of memory whose value is polled and displayed (see Section 3.1). More complex signals consist of functions that return a signal sampling point.

Gscope focuses on visualization of time-sensitive software applications. It can be used for visualizing time-dependent variables such as network bandwidth, latency, jitter, fill levels of buffers in a pipeline, CPU utilization, etc. We have implemented *gscope* and have been using it for the last two years. We have used it for visualizing and debugging various time-sensitive applications, including a CPU scheduler [19], a quality-adaptive streaming media player [14], a network traffic generator called *mxttraf* [13], and various control algorithms such as a software implementation of a phase-lock loop [9]. We believe that applications using *gscope* will see a direct benefit in terms of reducing the visualizing, debugging and testing cycle time.

Some of the key features of *gscope* are: support for multiple scopes and signals, dynamic addition and removal

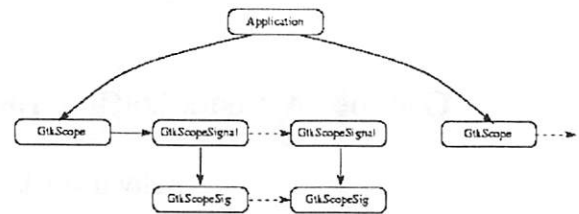


Figure 1: The GtkScope widget

of scopes and signals, adjustment of program or control parameters, support for arbitrary signal types, time and frequency representation of signals, support for discrete-time and event-driven signals, support for distributed visualization, saving of signal data, replay of signal data, adjustment of per-signal parameters and scope parameters, and a programmatic interface for every action that can be performed from the GUI. These features help fulfill the many of the goals that motivated the design of *gscope*.

In our experience, perhaps the most significant difference between the signals produced by software components and the signals typically visualized in an oscilloscope is the number of signal or event sources. Since software signals are not necessarily tied to specific pieces of hardware, applications can generate large numbers of disparate signals that need to be visualized and correlated. For instance, we use *gscope* to view dynamically changing process proportions as assigned by a CPU proportion-period scheduler [19]. Here, the number of signals depends on the number of running processes. As another example, since software signals are disconnected from hardware, they may be generated from remote sources (see Section 4.4).

The remainder of this paper describes *gscope* in more detail. Section 2 explains the *gscope* design by describing the graphical components of *gscope*. Section 3 presents key components of the interface that enable an application to communicate with *gscope*. Section 4 discusses various aspects of programming the *gscope* library and it describes some of our experiences with *gscope*. Section 5 examines related work in this area and Section 6 presents future directions for *Gscope*. Finally, Section 7 presents our conclusions.

2 Graphical Interface

Gscope is a graphical library and thus its various features are best explained by briefly describing the visual

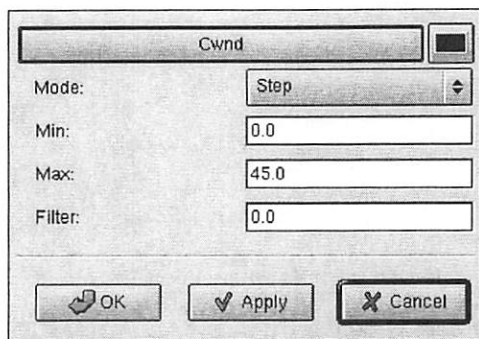


Figure 2: Signal Parameters Window

components of the library. Then this section explains how we have used *Gscope* to visualize TCP behavior in an experimental network. *Gscope* has been implemented using the Gnome [6] and GTK [3] graphical toolkits. These GUI toolkits are multi-platform although they are primarily designed for the X Window System. They are free software and part of the GNU Project [18]. Both Gnome and GTK use the *glib* library that provides generic system functionality independent of the GUI. For instance, *glib* provides portable support for event sources, threads, and file and socket I/O. *Gscope* uses some of this *glib* functionality.

The main graphical widget in the *gscope* library is called *GtkScope*, as shown in Figure 1. An application displays one or more signals by creating and passing a *GtkScopeSig* data structure for each signal to the library (see Section 3.1). The library creates a *GtkScopeSignal* object for each signal. Applications can create one or more *GtkScope* widgets.

A screen shot of the *GtkScope* widget with the embedded canvas displaying two signals is shown in Figure 4. The zoom and bias widgets below the canvas allow scaling and translating the signal data. The sampling period widget allows changing the polling period of the displayed data. The delay widget allows setting the delay with which buffered signals are displayed on the scope (further described in Section 3.1). The x-axis ruler is sized in seconds and the y-axis ruler has a scale from 0 to 100.

Under the zoom and bias widgets, signal parameters are displayed. Each signal has a signal name such as *CWND* and a *Value* button associated with it. Signal parameters, specified in the application using the *GtkScopeSig* data structure, can be modified by right clicking on the signal name, which brings up the window shown in Figure 2. Left clicking on the signal name toggles displaying the signal. When the *Value* button

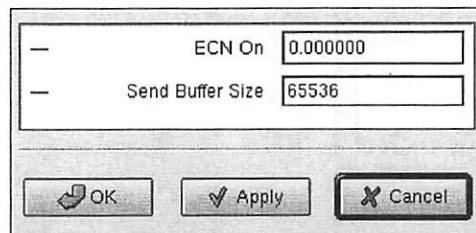


Figure 3: Application and Control Parameters Window

is pressed, the signal value is continuously displayed as shown with the *CWND* signal.

Gscope also allows storing, displaying and modifying application or control parameters that are application-wide and not specific to each *GtkScope* widget. Figure 3 shows an example of a control parameter window with two application parameters.

A Gscope Example

This section describes how the *gscope* library is used to visualize network behavior using the *mxttraf* network traffic generator application [13]. With *Mxttraf*, a small number of hosts can be used to saturate a network with a tunable mix of TCP and UDP traffic. The primary purpose of *mxttraf* is to allow stress testing of experimental networks.

The experiment shown in Figures 4 and 5 compares the behavior of TCP and ECN [8](explicit congestion notification) flows in a congested wide-area network. To emulate a simple wide-area network, we use a Linux router between a client and a server machine and use *nist-net* [17] to add delay and bandwidth constraints at the router. In this experiment, we use *mxttraf* to generate varying number of long-lived flows (called *elephants*) that transfer data from the server to the client. Figures 4 and 5 show two signals each. The *elephants* signal shows the number of long-lived flows over time. This number is changed from 8 to 16 roughly half way through the x-axis. The *CWND* signal shows the TCP or ECN congestion window (at the server) of one (arbitrarily chosen) long-lived flow in Figures 4 and 5 respectively. This window provides an estimate of the short-term bandwidth achieved by the flow.

These figures show how the window changes with a changing number of long-lived flows. While the absolute magnitude of the window is not very relevant in the short term period shown in the figures (since it changes

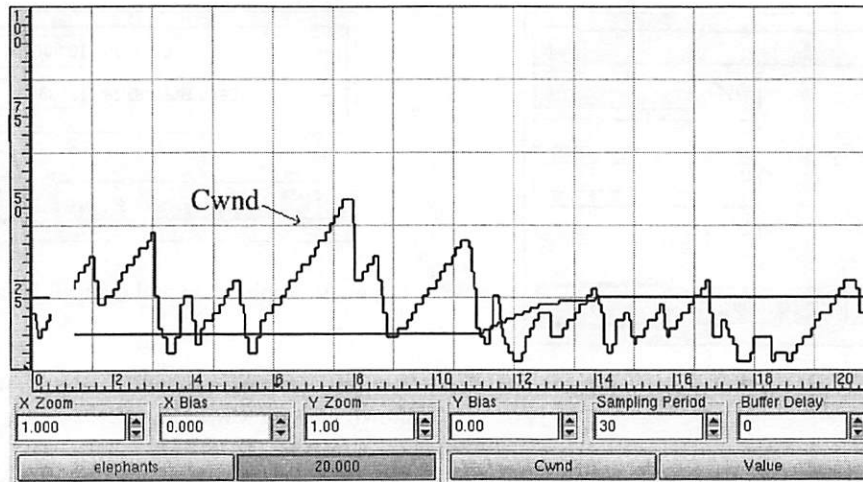


Figure 4: A snapshot of the GtkScope widget showing TCP behavior

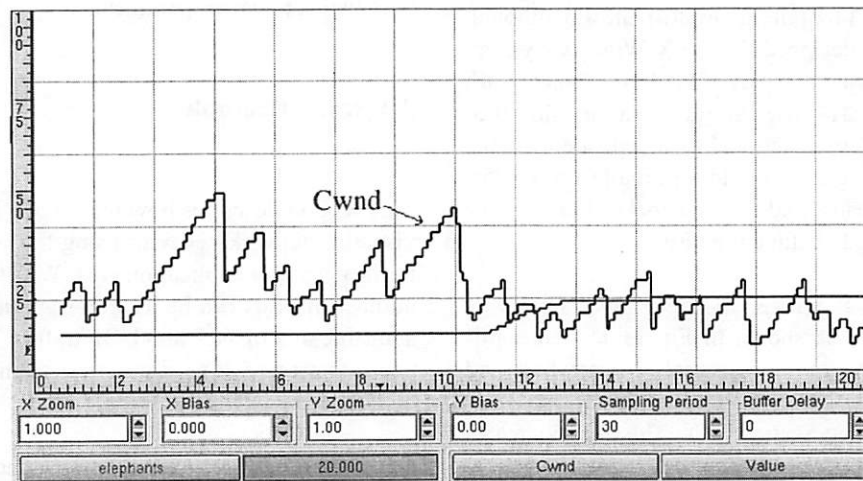


Figure 5: A snapshot of the GtkScope widget showing ECN behavior

dramatically over short intervals), one significant difference between the two flows is the number of timeouts experienced by each flow. Both TCP and ECN reduce the congestion window to one upon a timeout. The lowest value of the Cwnd signal in the graphs corresponds to a Cwnd value of one. The graphs show that while ECN does not hit this value, TCP hits it several times. Additional signals (not shown in the figures) confirm that there is a timeout each time Cwnd reaches one. Since timeouts affect TCP throughput and latency significantly, this experiment indicates that ECN can potentially improve flow throughput.

We use *mxtraf* to dynamically change the number of different types of flows, switch between different TCP variants and visualize network behavior in real time.

Such visualization has revealed several interesting properties (and bugs) in TCP behavior that would have been hard to determine otherwise. For instance, a TCP variant that we have implemented for low-latency TCP streaming [10] initially showed significant unexpected timeouts that we finally traced to an interaction with the SACK implementation.

3 Gscope API

This section describes the interface data structures that enable an application to communicate with *gscope*. The *gscope* library The *gscope* interface is relatively sim-

ple but powerful. The *gscope* interface consists of three components: 1) signal specification, 2) control parameter specification for configuring the application and 3) tuple format for streaming, recording and viewing data.

3.1 Signal Interface

Gscope can acquire signal data from applications in one of two acquisition modes: *polling* or *playback*. In polling mode, signals are obtained from the running program using the signal interface described below. Polled signals can be *unbuffered* or *buffered*. In unbuffered mode, *gscope* polls and displays single sampling points. In buffered mode, applications enqueue signal samples with timestamps into a buffer and *gscope* displays these samples with a user-specified delay. The buffered mode enables applications to push data to the scope. For instance, an application can listen for kernel events on a *netlink* socket and push these event samples to the *gscope* buffer. *Gscope* polls the buffer periodically to display the samples. Polled signals can be displayed in the time or frequency domain. In addition, the polled data can be recorded to a file. Section 4 discusses the polling overhead and the finest polling granularity that is supported in *gscope*.

In the playback mode, data is obtained from a file and displayed. This file format is described in Section 3.3. Both polling and playback modes have a polling period associated with them. In both modes, data is displayed one pixel apart each polling period (for the default zoom value).

A signal is specified to the *gscope* library using a *GtkScopeSig* structure shown below:

```
typedef struct {
    char          *name;      /* signal name */
    GtkScopeSigData signal; /* signal data */
    /* color, min, max, line, hidden, filter */
} GtkScopeSig;
```

The name is the name of the signal and the *signal* field is used to obtain signal data. This field is described with an example below. The rest of the fields are optional parameters that specify the color of the signal, the minimum and maximum value of the signal displayed (for default zoom and bias values), the line mode in which the signal is displayed, whether the signal is hidden or visible, and a parameter α for low-pass filtering the signal. The low-pass filter uses the following equation to filter the signal: $y_i = \alpha y_{i-1} + (1 - \alpha)x_i$. Here,

x_i is the signal point and y_i is the filtered signal point. The α filter parameter ranges from the default value of zero (unfiltered signal) to one.

The examples below show the *GtkScopeSig* specification for the *elephants* and *CWND* signals. The *elephants* signal consists of an integer value that will be sampled by *gscope*. The *CWND* signal uses the *get_cwnd* function to determine the *CWND* value of the socket *fd*.

```
int elephants;
GtkScopeSig elephants_sig = {
    name: "elephants",
    signal: {type: INTEGER, {i: &elephants}},
    min: 0, max: 40 /* optional */
};

int fd; /* socket file descriptor */
GtkScopeSig cwnd_sig = {
    name: "Cwnd",
    signal: {type: FUNC, {fn: {get_cwnd, fd}}},
};
```

The signal can be of type *INTEGER*, *BOOLEAN*, *SHORT*, *FLOAT*, *FUNC* or *BUFFER* and this type determines how signals are sampled. When the signal type is *BUFFER*, the signal is buffered, otherwise it is unbuffered.

Unbuffered Signals For unbuffered signals, the *INTEGER*, *BOOLEAN*, etc. field is sampled and displayed. of the union in the *GtkScopeSigData* structure depending on the type of the signal. When the signal type is *FUNC*, the function is invoked with the two arguments *arg1* and *arg2* (passed in by the user during *GtkScopeSig* initialization) and the function's return value is the value of the signal data. The function mechanism allows reading arbitrary signal data.

Buffered Signals For buffered signals, *gscope* reads data from a scope-wide buffer that has timestamped signal data in a tuple format (described in Section 3.3) and displays this data with a user-specified delay. *Gscope* provides applications an API for inserting the timestamped signal data in the buffer.

3.2 Control Parameter Interface

Application or control parameters as shown in Figure 3 can be read and modified by the *gscope* library using the `GtkScopeParameter` structure. These parameters are not displayed but generally used to modify application behavior. The `GtkScopeParameter` structure is very similar to the `GtkScopeSig` structure. However, while signals can only be read, application parameters can be read and written also.

3.3 Tuple Format

Signals can be streamed to *gscope*. For instance, streamed signals allow distributed visualization in real time. Signals can also be recorded to a file and *gscope* can replay signals from the file. In all these cases, signal data is delivered, generated or stored in a textual tuple format. Each tuple consists of three quantities: *time*, *value* and *signal name*. This format allows multiple signals to be delivered to *gscope* or recorded in the same file. As a special case, if there is only one signal, then the third quantity may not exist. In that case, signals are simply time-value tuples.

When signals are streamed or replayed from a recorded file, the time field of successive tuples is in increasing time order and its value is in milliseconds. Data is displayed one pixel apart for each polling period (for the default zoom value). For instance, if the polling period is 50 ms, then data points in the file that are 100 ms apart will be displayed 2 pixels apart.

3.4 Programming With Gscope

The *Gscope* library has a programmatic interface for every action that can be performed from the GUI. Figure 6 presents a fragment of a simple program that shows how the *gscope* library is used. After creating the scope, the `elephants_sig` signal (defined in Section 3.1) is added to `scope` and `scope` is set to polling mode, where it polls the value of `elephants` every 50 ms. The function that manipulates the `elephants` value is called `read_program` and it runs when the server has control data available from the client. In this usage style, the `read_program` function is I/O driven and performs non-blocking calls. Other ways of using the *gscope* library include 1) periodic invocation of `read_program` and 2) separation of the scope into its

```
main()
{
    ...
    scope = gtk_scope_new(name, width, height);
    /* sig defined in Section 3.1 */
    gtk_scope_signal_new(scope, elephants_sig);
    /* sampling period is 50 ms */
    gtk_scope_set_polling_mode(scope, 50);
    /* set polling to start state */
    gtk_scope_start_polling(scope);
    /* register read_program with I/O loop */
    g_io_add_watch(..., G_IO_IN, read_program, fd);
    /* main loop: calls read_program when fd
       has input data */
    gtk_main(); /* doesn't return */
}

gint
read_program(int fd)
{
    control_info = read_control_info(fd);
    if (elephants != control_info.elephants) {
        start or stop elephants;
        /* change signal value */
        elephants = control_info.elephants;
    }
    return TRUE;
}
```

Figure 6: A sample *gscope* program

own thread. These issues are discussed further in Section 4.3.

4 Discussion

The previous section has described the *gscope* API and how the *gscope* library can be used. This section discusses various aspects of programming the *gscope* library in more detail and it describes some of our experiences with *gscope*. Section 4.1 describes portability issues with the *gscope* library. Section 4.2 examines how *gscope* can be used for different types of signals effectively. Section 4.3 describes when it is appropriate to have a single-threaded or a multi-threaded *gscope* application, while Section 4.4 describes how data is polled and displayed from a distributed application. Section 4.5 describes the polling granularity in the current implementation and thus the type and range of applications that can be supported. Finally, Section 4.6 discusses the overhead of our approach.

4.1 Implementation Portability

Gscope has been implemented on the Linux OS and we have been using it for the last two years. *Gscope* can be installed on a vanilla Linux system that has Gnome software installed on it. Although, *Gscope* has not been ported to other free source operating systems such as BSD, we believe that the porting process should be simple since *Gscope* does not use any Linux specific functionality and Gnome has been ported to other OSs.

4.2 Signal Types

Applications often produce various types of signal data, such as clocked signals and event-driven signals. For instance, bandwidth monitoring can be done based on events that are packet arrivals. *Gscope* implements a discrete-time polling system but can also handle event-driven signal generation. Below, we describe various signal types and how *gscope* handles them.

Sample and Hold Applications can be designed so that certain events change a state and then the state is held until the next event changes the state. Between event arrivals, polling can detect the previous event by monitoring the held state. For instance, the state can be the end-to-end packet latency that can change on each packet arrival event. If the polling frequency is sufficiently high, all packet arrival events can be captured. This approach requires knowing the shortest period of back-to-back event arrival.

Periodic Signals Signals can be periodic, in which case, such signals can be viewed by polling at the same period. For instance, we use *gscope* to view dynamically changing process proportions as assigned by a real-rate proportion-period scheduler [19]. These proportions are assigned at the granularity of the process period and we set the scope polling period to be same as the process period. This approach does not require phase alignment between process period and the polling period since the signal is held between process periods.

Buffering Events can be buffered and then polling can display data with some delay. For instance, a device driver could poll a memory mapped device at the appropriate frequency and queue data to a buffer that is then polled by *gscope*. In *gscope*, the buffering interface is implemented with buffered signals

described in Section 3.1. This approach may seem like cheating since one of the main purposes of the scope is to poll the data directly. However, decoupling the data collection from the data display has several benefits. For instance, data can be collected and displayed on different machines, thus allowing distributed or client-server application data visualization as discussed in Section 4.4. In addition, data can be captured only when certain conditions are triggered, i.e. at “interesting” times. At other times, data collection and display would have little or no overhead. A similar trigger-driven sampling approach is used by hardware oscilloscopes.

Event Aggregation Another very effective method for visualizing event-driven signals is event aggregation. In this method, event data is aggregated between polling intervals and then displayed. For instance, applications may want to display the maximum value of an event sample between polling intervals. An example of using the maximum sample value is to display the maximum latency of a network connection. Rather than displaying the latency of each packet (as discussed in Sample and Hold) or the maximum latency over the life time of the connection, it may be useful to display the maximum latency within each polling interval. *Gscope* provides aggregation functions shown below that aggregate data between polling intervals. Examples for network connections are described for each function.

Maximum and Minimum maximum and minimum sample, e.g., latency.

Sum Sum of the sample values, e.g., bytes received.

Rate Ratio of the sum of sample values to the polling period, e.g., bandwidth in bytes per second.

Average Ratio of the sum of sample values to the number of events, e.g., bytes per packet.

Events Number of events, e.g., number of packets.

AnyEvent Did an event occur between polling intervals, e.g., any packet arrived?

4.3 Single vs. Multi-Threaded Applications

Gscope is thread-safe and can be used by both single-threaded and multi-threaded applications. With multi-threaded applications, typically *Gscope* is run in its own thread while the application that is generating signals

is run in a separate thread. This approach allows the *gscope* GUI to be scheduled independently of the application (unless *gscope* signals make application calls that need to acquire locks). However, it is the application thread's responsibility to acquire a global GTK lock if it needs to make *gscope* API calls.

Single-threaded *gscope* applications must use event-driven programming. Such applications should either be periodic or they should be I/O driven and they should use non-blocking I/O system calls (since blocking calls would block the GUI as well). Periodic applications are supported directly by *gscope*. I/O driven applications can use the GTK `GIOChannel` functions to drive their events as shown in Figure 6. This approach allows all GUI and application events to be handled by the same event loop and does not require any locking. However, application logic can become more complex due to the use of non-blocking I/O system calls. We have implemented a single-threaded I/O driven *gscope* client-server library that is described in the next section.

4.4 Distributed Applications

Gscope supports monitoring and visualization of distributed applications. It implements a single-threaded I/O driven client-server library that can be used by applications to monitor remote data. Clients use the *gscope* client API to connect to a server that uses the *gscope* server library. Clients asynchronously send `BUFFER` signal data in tuple format (described in Section 3.3) to the server. The server receives data from one or more clients asynchronously and buffers the data. It then displays these `BUFFER` signals to one or more scopes with a user-specified delay as described in Section 3.1. Data arriving at the server after this delay is not buffered but dropped immediately.

Currently, we use the *gscope* client-server library in the `mxttraf` network traffic generator. The *gscope* client-server library allows visualizing and correlating client, server and network behavior (connections per second, connection errors per second, network throughput, latency, etc.) within a single scope.

4.5 Polling Granularity

Gscope uses the GTK timeout mechanism to implement polling. The default GTK timeout implementation uses the timeout feature of the POSIX `select` call. Although `select` allows specifying the timeout with a

microsecond granularity, typically the kernel wakes processes at the granularity of the normal timer interrupt. The timer interrupt generally has a much coarser granularity. For instance, on Linux, this granularity is 10 ms. Thus *gscope*, which is implemented on Linux, is currently limited to this polling interval and has a maximum frequency is 100 Hz.²

In addition to coarse granularity timeouts, scheduling latencies in the kernel can induce loss in polling timeouts under heavy loads. To handle this problem, *Gscope* keeps track of lost timeouts and advances the scope refresh appropriately.

Compared to an oscilloscope, *Gscope* has a much coarser polling granularity and thus relatively low bandwidth. For instance, the current *Gscope* implementation would not be appropriate for real-time low-delay display of a speech recognition application that monitors phone-line quality 8 KHz audio signals. Fortunately, in our experience, many software applications don't have tight polling requirements. Coarse granularity polling works well for three reasons: 1) many software applications have coarse time scales, 2) debugging software applications often only requires visualizing the long term trends of the signal, 3) many applications generate event-driven signals that are handled by techniques such as buffering or event aggregation as explained in Section 4.2. For instance, the audio signal could be read from the audio device and buffered by an application and *gscope* can display the signal with some delay using buffered signals. In our experience, the 10 ms polling granularity and loss of polling timeouts has not been a limiting factor for the *gscope* applications that we have implemented. However, Section 6 describes some directions for improving the polling granularity.

4.6 Scope Overhead

We measured the overhead of using the *gscope* library by running a simple application that polls and displays several different integer values. To measure overhead, we use a CPU load program that runs in a tight loop at a low priority and measures the number of loop iterations it can perform at any given period. The ratio of the iteration count when running *gscope* versus on an idle system gives an estimate of the *gscope* overhead.

The *gscope* CPU overhead on a 600 MHz Pentium III processor is less than two percent while polling at 10 ms granularity (smallest granularity supported by the sys-

²The `setitimer` periodic timer call behaves similarly.

tem) and less than one percent at 50 ms granularity. The increase in overhead with increasing number of signals being displayed ranges from 0.02 to 0.05 percent per signal. When compared to the number of signals displayed, polling granularity has a much larger effect on CPU consumption.

5 Related Work

This section provides some background on oscilloscopes and compares them with *gscope*. Then it describes some oscilloscope-like applications that have been developed in the free software community.

The oscilloscope is essentially a graph-displaying device – it draws a graph of an electrical signal. Oscilloscopes can help determine various signal properties: time and voltage values of a signal, frequency of an oscillating signal, phase difference between two oscillating signals, a malfunctioning component that is distorting the signal, AC and DC components of a signal and noise in a signal. Oscilloscopes put significant effort on visualization of repeating waveforms. For instance, they have trigger controls that help stabilize such waveforms. Oscilloscopes can be analog or digital. Analog oscilloscopes are preferred when it is important to display rapidly varying signals in “real time”. However, digital oscilloscopes allow capture and viewing of events that may happen only once. They can process the digital waveform data or send the data to a computer for processing. Like *gscope*, they can store the digital waveform data for later viewing and printing and they also allow event aggregation.

Gscope is similar in functionality to *gstripchart* [12], the Gnome stripchart program, that charts various user-specified parameters as a function of time such as CPU load and network traffic levels. The *gstripchart* program periodically reads data from a file, extracts a value and displays these values. However, unlike *Gscope*, *gstripchart* has a configuration file based interface rather than a programmatic interface, which limits its use for debugging or modifying system behavior.

There is large body of work related to implementing software digital oscilloscope functionality for audio visualization. The basic idea is to record sound with a microphone and then display the digitized sound waves. *Xoscope* [20] is one such program. *Xmms* [4] displays sound frequency during audio playback. *Baudline* [2]

is a real-time signal analysis tool and an offline time-frequency browser. These programs emulate the functionality of a digital oscilloscope much more closely than *gscope*. However, these programs are focusing on audio visualization while *gscope* focuses on visualization and debugging of software behavior. Thus certain oscilloscope features are not appropriate for *gscope* and vice-versa.

There are hundreds of measurement tools that can be used for capturing system and network performance [1]. *Gscope* complements them because it can be used to visualize their output in real-time.

6 Future Work

We expect to see more integration of oscilloscope functionality in *gscope*. *Gscope* currently does not have support for repeating waveforms. Thus, many oscilloscope features such as triggers that stabilize repeating waveforms or waveform envelop generation are not implemented in *gscope*. *Gscope* does not currently support printing of recorded data. Also, it does not have bindings for languages other than C.

There are several options for supporting applications with more stringent polling requirements. First, Linux exposes the real-time clock on the Intel x86 processor that can generate interrupts at a maximum frequency of 8KHz. Unfortunately, this clock is exposed only to processes with `root` privileges and can only be used by one application at a time. Further, it is not clear how this mechanism can be used together with the GTK polling or event handling mechanism. The benefit of using the GTK polling mechanism is that all events, GUI as well as application events, are handled by the same mechanism and this allows implementing fully event-driven applications. Such an implementation is the norm for GUI applications [16, 7].³

A second option is to improve the granularity of `select` in the kernel by connecting it to a more general fine-grained timing facility, such as soft-timers [5]. Improving the granularity of `select` will automatically help to improve the granularity of the GTK polling mechanism. Finally, kernel scheduling latencies can be reduced by using a preemptive kernel [15].

³Gimp is event-driven but uses a thread per processor in an SMP environment to optimize certain types of processing.

7 Conclusions

Gscope is designed for visualizing time-sensitive software applications. Its goal is to reduce the cycle time needed for visualizing, testing and debugging time-sensitive applications by providing an oscilloscope-like interface that can be integrated with the application. In this paper, we have described the design and interface of the *gscope* library, presented some simple examples of using the library and then discussed various aspects related to programming the library. We have used *gscope* successfully in many of our applications and have built several compelling demos of our research work using this library. *Gscope* is free software. More information about *gscope* is available at <http://gscope.sf.net>.

References

- [1] NLANR Network Performance and Measurement Tools. <http://dast.nlanr.net/npmt/>.
- [2] The Baudline Real-Time Signal Analysis Tool. <http://www.baudline.com>.
- [3] The GTK Graphical User Interface Toolkit. <http://www.gtk.org>.
- [4] Peter Alm, Thomas Nilsson, and et al. XMMS: A Cross Platform Multimedia Player. <http://www.xmms.org>.
- [5] Mohin Aron and Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems*, August 2000.
- [6] Miguel de Icaza and et al. The Gnome Desktop Environment. <http://www.gnome.org>.
- [7] Miguel de Icaza and et al. The Gnumeric Spreadsheet. <http://www.gnome.org/projects/gnumeric>.
- [8] Sally Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):10–23, 1994.
- [9] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, third edition, 1997.
- [10] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting Low Latency TCP-Based Media Streams. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS)*, May 2002. To appear.
- [11] Ramesh Jain. TeleExperience: Communicating Compelling Experiences. Keynote speech at ACM Multimedia 2001.
- [12] John Kodis. Gstripchart: a Stripchart-Like Plotting Program. <http://users.jagunet.com/~kodis/gstripchart/gstripchart.html>.
- [13] Charles Krasic. The Mxtraf Traffic Generator. <http://sourceforge.net/projects/mxtraf>.
- [14] Charles Krasic, Kang Li, and Jonathan Walpole. The Case for Streaming Multimedia with TCP. In *8th International Workshop on Interactive Distributed Multimedia Systems (iDMS 2001)*, pages 213–218, Sep 2001. Lancaster, UK.
- [15] Robert Love. The Linux Kernel Preemption Project. <http://kpreempt.sourceforge.net>.
- [16] Peter Mattis and Spencer Kimball. Gimp, the GNU Image Manipulation Program. <http://www.gimp.org>.
- [17] NIST. The NIST Network Emulation Tool. <http://www.antd.nist.gov/itg/nistnet>.
- [18] Richard M. Stallman and et al. The GNU Project. <http://www.gnu.org>.
- [19] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*. USENIX, February 1999.
- [20] Timothy D. Witham. Xoscope: a Digital Oscilloscope for Linux. <http://xoscope.sourceforge.net>.

Inferring Scheduling Behavior with Hourglass

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

<http://www.cs.utah.edu/~regehr>

Abstract

Although computer programs explicitly represent data values, time values are usually implicit. This makes it difficult to analyze and debug *real-time* programs whose correctness depends partially on the time at which results are computed. This paper shows how to use Hourglass, an instrumented, synthetic real-time application, to make inferences about what is happening on a computer at millisecond and microsecond granularities. These inferences are possible because Hourglass records a very fine-grained map of when each of its threads runs, and because Hourglass supports a variety of *thread execution models* that model the properties and requirements of non-synthetic real-time applications. We conclude that between measurements and inferences, surprisingly detailed knowledge about scheduling behavior can be obtained without modifying, or even explicitly interacting with, the operating system kernel.

1 Introduction

Real-time applications such as games, audio playback, video display, and voice recognition need to finish computations by certain times, in addition to producing the right result, in order to operate correctly. Task execution can become complex because tasks share resources, because time-sharing schedulers such as the ones in Linux and FreeBSD execute complex algorithms, and because kernel activity such as hardware and software interrupt handlers can interfere with application execution. These factors often make it difficult to figure out what is really happening at microsecond and millisecond granularities when a real-time application runs. The situation is complicated by the many modified Linuxes that provide improved real-time services, since each has different performance characteristics. These real-time enhanced kernels fall into two main categories. First, there are those that improve the real-time performance of basic mechanisms, such as high-resolution timers [1], Robert Love's

preemptible kernel patch [12], Andrew Morton's lock-breaking patch [13], and the TimeSys Linux/GPL kernel [23], which includes a number of scheduler enhancements and also makes most device driver activity preemptible by threads. Second, there are those that change the CPU scheduling algorithm, such as Linux-SRT [4], QLinux [22], RED-Linux [25], and Linux/RK [14].

It is difficult to measure, analyze, and understand the behavior of real-time applications that are scheduled by these different Linux variants and by other operating systems. To address this problem we have developed *Hourglass*, a heavily instrumented synthetic real-time application that operates entirely in user space and requires no modifications to the operating system. Our focus is on Linux, but Hourglass also runs on Windows 2000 and FreeBSD; it should be easy to port to other Win32- and Unix-based systems.

2 Monitoring Scheduling Behavior

Broadly speaking, there are two ways to learn about the run-time behavior of an application: by instrumenting the application and by instrumenting the kernel. Each approach has advantages and disadvantages.

2.1 Instrumented Kernels

Kernel-based instrumentation and measurement techniques can be divided into approaches that monitor the execution of a single application and those that monitor the system as a whole. An example that fits into the first category is `ptrace()`, a system facility that permits a parent process to monitor and intercept any kernel calls made by a child process. The second category is exemplified by the Linux Trace Toolkit (LTT) [26], a general-purpose event logging facility for the Linux kernel. The important properties of LTT are that it logs events into physical memory, doing no I/O until requested, and that it does not add a lot of overhead to normal operations.

Some of the factors favoring use of an instrumented kernel include:

- Since the kernel is not treated as a black box, it is possible to figure out exactly why particular timing effects were observed rather than relying on indirect inferences.
- Since applications do not need to be modified, real applications can be run without adding instrumentation code that may increase complexity or affect their timing behavior.

2.2 Instrumented Applications

Instrumented applications fall into two categories. First, it is possible to add instrumentation to real applications and second, synthetic applications can be used. Real applications can be instrumented in a variety of ways: by hand, by linking against instrumented library routines, by interposing on library calls using macro or linker tricks, or by rewriting a binary. For example, *gscope* [6] is a visualization tool that uses the metaphor of an oscilloscope; it provides an API that applications can use to send signals to *gscope*, where they are processed and displayed. *ATOM* [21] exemplifies a very different approach: it uses binary rewriting to add customized instrumentation to an application. Because *ATOM* provides very fine-grained instrumentation it could be used to add timing instrumentation to real applications, although care would have to be taken to avoid slowing a program down too much.

Synthetic applications such as *Hourglass* need not be instrumented further because their entire purpose is to provide instrumentation. By abstracting away from real applications it is possible to obtain very predictable and controllable behavior, to get fine-grained timing measurements, and to easily model a wide variety of application scenarios without changing complex application code. On the other hand some application characteristics, such as patterns of synchronization between threads, may be difficult or impossible to model using a synthetic application.

Instrumented applications have the following benefits:

- The timing information reported is guaranteed to be authentic in the sense that actual application scheduling behavior is measured. This is often better than measuring timing in the kernel, where it may be difficult to instrument all code of interest, e.g. interrupt handlers, bottom-half handlers, etc. Furthermore, cache effects can be measured at the application level.

```
thrd_func () {
    sleep_until (my_start_time);
    while (now < my_finish_time) {
        run_workload ()
    }
}

main () {
    parse_command_line_args ()
    foreach (0..num_threads-1) {
        pthread_create (thrd_func)
    }
    sleep_until (overall_finish_time)
    print_results ()
}
```

Figure 1: Hourglass in a nutshell

- Since the kernel is not modified, there is no risk of destabilizing the system, or breaking or slowing down non-real-time applications. Also, there are no kernel patches to conflict with the many modified Linux kernels listed in Section 1.

2.3 Summary

Kernel-based instrumentation is the best way to measure OS-specific metrics such as the time to execute a particular code path in the kernel. A combination of kernel-based instrumentation and hand-inserted application instrumentation is almost always the right choice for debugging a specific real-time application on a specific operating system. On the other hand, for performing a comparative evaluation of real-time operating systems or for exploring application scenarios other than those provided by available applications, a user-space synthetic application is almost always the right choice.

3 Hourglass Structure and Internals

Hourglass is structured as a collection of cooperating threads: one for each thread that the user requests, plus the thread that initially runs *main()*. Figure 1 shows pseudocode for *Hourglass*. Most of the complexity is in *run_workload()*, which is covered in this section.

3.1 Detecting Gaps

While starting up, *Hourglass* allocates a memory buffer (about 5 MB long, by default) for storing an *execution trace*. The trace is generated by threads as they run: each time a thread detects a gap in its execution, it allocates a record in the trace buffer and uses it to store the start and end time of a continuous block of CPU time that the thread received. Taken together, these records produce a

```

while (1) {
    now = rdtsc ()
    if (now - last_exec > GAP) {
        rec = allocate_record ()
        rec.id = my_pid
        rec.start = exec_start
        rec.end = last_exec
        exec_start = now
    }
    last_exec = now
}

```

Figure 2: Algorithm for recording intervals of continuous CPU time received by a thread

map of how the CPU was used during a particular Hourglass run.

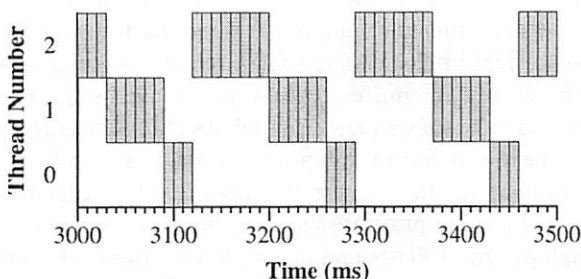
To learn when it is receiving CPU time, each thread continuously polls the Pentium timestamp counter by executing the `rdtsc` instruction, which returns a 64-bit quantity indicating the number of processor cycles since the machine was booted. If the difference between successive reads of the timestamp counter exceeds a `GAP` threshold, the thread considers its execution to have been interrupted and adds an element to the trace buffer recording the start and finish of the time interval during which it had uninterrupted access to the processor. By setting `GAP` appropriately, even very small interruptions, such as those caused by interrupt handlers that run for a few microseconds, can be detected. Figure 2 shows pseudocode for the gap-detection algorithm, and Figure 3 shows some execution traces produced by post-processing Hourglass output with `render_trace`, a script that is distributed with Hourglass. Gray rectangles represent continuous blocks of CPU time allocated to a thread. To produce these traces, Hourglass was run with this command line

```

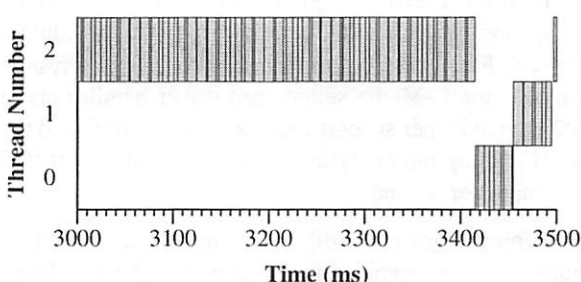
hourglass -n 3 -d 5s \
  -t 0 -p LOW \
  -t 1 -p NORMAL \
  -t 2 -p HIGH

```

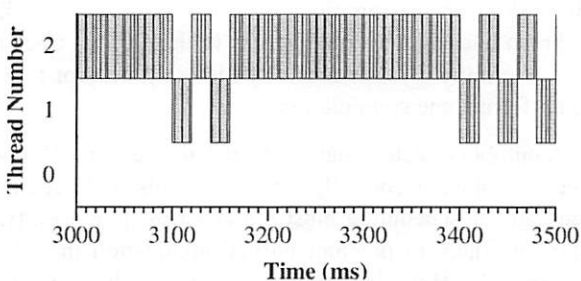
which creates three threads at different priorities and lets them record an execution trace for five seconds (Appendix A provides command line usage information for Hourglass). The top three traces in Figure 3 are 500 ms segments of the 5 s traces, and were respectively taken on an otherwise idle Linux machine, an otherwise idle Windows 2000 machine, and an otherwise idle FreeBSD machine. The bottom trace segment is 100 ms long in order to show more detail, and was taken on a Linux machine running `xmms`, an audio player application. The operating system versions used are described at the beginning of Section 5.



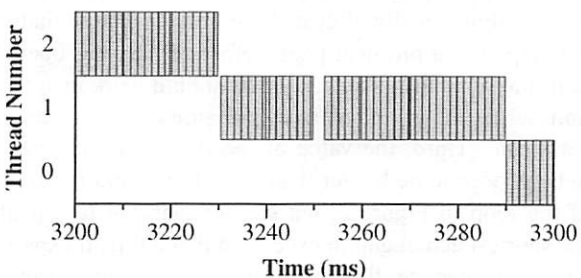
(a) Hourglass running by itself on plain Linux 2.4.17



(b) Hourglass running by itself on Windows 2000



(c) Hourglass running by itself on FreeBSD 4.5



(d) Detail of Hourglass sharing the processor with `xmms` on plain Linux 2.4.17

Figure 3: Example execution traces

The first trace clearly shows Linux's clock interrupts arriving every 10 ms, and it also shows that higher-priority threads run for longer than low-priority threads rather than running more often. Running for longer is the correct implementation for a server operating system since it minimizes context switch overhead. The second trace shows that the Windows 2000 scheduler is not nearly as fair to low-priority threads as the Linux scheduler is. In fact, the Windows 2000 scheduler is almost a static priority scheduler: in the presence of a high-priority CPU-bound thread, lower-priority threads only receive brief use of the CPU every few seconds [20, pp. 367–368]. The FreeBSD trace (third from the top) seems to show some sort of complex behavior caused by its multi-level feedback queue. FreeBSD lies between Linux and Windows 2000 with respect to scheduling fairness. Finally, the bottom trace is broken up by system activity: the C-Media sound card driver handles about 700 interrupts per second when xmms is active, and the act of reading mp3 data from disk causes about 20 IDE interrupts per second.

Keeping close track of when each thread runs provides a basis for implementing many useful capabilities. For example, by examining gaps in thread execution, we can determine how long various events of interest such as context switches and interrupts take to execute. By adding up the durations of intervals during which a thread had uninterrupted access to the CPU, a thread can determine if it has received enough processor time to perform some computation.

A number of details have to be taken care of for Hourglass to operate correctly. First, threads that require low-latency scheduling must run at a real-time priority, allowing them to preempt normal application threads. Second, the Hourglass process should be locked into physical memory in order to avoid reporting spurious delays that are caused by page faults. On systems that do not support `mlockall()`, Hourglass touches each page of dynamically allocated memory to ensure that it is mapped to a physical page before any timing operations are run. This is a hack, but it should work in situations where there is no memory pressure during an Hourglass run. Third, the value of the GAP constant needs to be chosen to be longer than the typical iteration time of the loop in Figure 2, but shorter than the length of the shortest actual gap in execution that a thread experiences. In practice, this is not difficult: on our test machine, an 850 MHz Pentium III, the execution trace loop usually takes about 225 ns to execute. Setting GAP to twice this value appears to avoid detecting spurious gaps while still detecting even fast interrupt handlers. Fourth, since `rdtsc` returns values in cycles, Hourglass needs to be aware of the clock speed of the processor it is run-

ning on. This can be determined by comparing the rates at which `rdtsc` and `gettimeofday()` run.

3.2 Thread Execution Models

To facilitate the modeling of a variety of application scenarios, Hourglass supports a number of thread execution models. All thread models except the last, the latency test, record records of when they run in order to participate in the creation of an execution trace.

CPU bound: These threads simply run whenever they can.

CPU bound, scanning: These threads are CPU-bound, and they sequentially access elements in an array, modeling tasks with non-zero working sets in the data caches.

CPU bound, yielding: These threads are CPU-bound but voluntarily yield the processor each time they have received a configurable amount of CPU time.

Periodic: These threads have a characteristic execution time and period. During each period they run until they have received their execution time, at which point they block until the beginning of the next period. The timer that they block on can be selected at run time. Each time a periodic thread has not received its execution time by the end of its period, it registers a deadline miss. Periodic threads model real-time applications that have a characteristic rate, such as real-time audio processing, video decoding, or a software modem.

CPU bound, periodic: These threads have a characteristic execution time and period, but they never block. Rather, each time a thread receives its CPU time requirement, it simply begins another period. This models the class of applications such as games and other real-time simulations that must provide some minimum frame rate, but can opportunistically use extra CPU time to provide higher frame rates. For example, we might require that a game produce at least 20 frames per second. If it requires 10 ms to produce a single frame, then its execution time is 10 ms and its period is 50 ms. This thread will register a deadline miss any time a 50 ms period elapses where no frame is produced. If all CPU time is available to this thread, it will provide 100 frames per second.

Latency test: When these threads run they record the current time and then sleep until a period has passed. They are used to measure dispatch latency: the time between when a thread becomes ready and when it starts to run. Latency test threads are different from periodic threads in that they automatically synchronize their periods to the timer source rather than maintaining an independent period. In other words, each time they awaken

they compute their next wakeup time by adding their period to the current time instead of the time at which they should have awakened. This simplifies latency testing by eliminating phasing errors that can make things difficult for actual real-time applications.

It is easy to add new thread execution models to Hourglass. For example, it might be useful to add a new periodic task model where, instead of requiring the same amount of CPU time during each period, the execution time requirement follows a statistical distribution or uses a table-driven approach to simulate the deterministic variability shown by an MPEG-2 video decoder as it processes different types of frames.

3.3 Time and Timers

In most general-purpose operating systems it is hard to gain control of the CPU at a precise time, unless that time happens to be synchronized with a periodic clock interrupt. In many systems clock interrupts arrive every 10 ms. In Linux and Windows 2000, better timing can be obtained using the real-time clock (RTC), which can be programmed to interrupt the CPU at power-of-two frequencies; for example, at 1024 Hz it provides 976 μ s resolution — precise enough for most applications. Even better timers for Linux can be obtained through a kernel patch [1] that provides the POSIX timer interface to applications based on either the RTC, the programmable interrupt timer (PIT), or the ACPI power management timer. The last two can achieve small-microsecond accuracy. Windows 2000 provides *multimedia timers* that, on most platforms, provide a timer resolution of about 1 ms. Hourglass is capable of using all of these timers, and even of mixing them within a run.

3.4 Data Output and Analysis

Hourglass has three main design goals with respect to its output. First, all output happens after a run finishes, in order to avoid contaminating timing information with I/O effects. Second, Hourglass produces output that is as complete as possible, and is (in principle) human-readable. And finally, lines that contain data that is likely to be of interest to scripts are tagged with special strings that make them easy to identify.

4 Inferring Scheduling Behavior

Because Hourglass runs entirely in user space, the causes for gaps need to be inferred, and often this knowledge can only be statistical. The following list provides examples of some kinds of inferences that can be made using Hourglass, and it illustrates the distinction between statistical and non-statistical inference.

- If Thread 1 runs, a gap occurs, and then Thread 2 runs, one can make the non-statistical inference that a context switch occurred during the gap. Statistical inferences that can be made after looking at a number of such gaps include: (1) that the shortest gap indicates the fastest path through the context switch code, (2) that if there are many gaps of about the same length, and this length isn't much longer than the shortest gap, then the median member of this set of gaps is representative of the expected context switch time, and (3) that any very long outlier gaps have either been contaminated by other system activity such as interrupt handlers, or that the context switch code has an unpredictable execution time.
- If a context switch is observed between two threads that are known to never yield or block (voluntarily or involuntarily), one can make a non-statistical inference that the context switch was an involuntary preemption, and hence a clock interrupt must have occurred during the gap, in addition to a context switch.
- If the highest priority active thread is known to never block, gaps in its execution must be caused by high-priority non-thread activity such as interrupts and bottom-half device driver routines.
- If threads running on an idle machine have few gaps, and if threads running on the same machine under a particular workload (e.g. receiving network data) show many gaps, one can make the statistical inference that the workload is causing the gaps.
- If a thread running on an otherwise idle machine has occasional high dispatch latencies when using one kind of timer, but not another, then the statistical inference can be made that one of the timer implementations is inaccurate or broken. One can have higher confidence in this inference when running on a low-latency or preemptible kernel that almost never shows long "real" gaps.

If these kinds of inferences are insufficient, there are two potential solutions. First, it might be possible that a more clever use of Hourglass, or a modification to Hourglass, would permit stronger inferences to be made. Second, it might be necessary to stop viewing the kernel as a black-box, and to start using a package such as the Linux Trace Toolkit. Although Hourglass does not currently interact with LTT, it would be useful to be able to correlate results from the two sources. This would be straightforward because a single time line, from `rdtsc`, is available.

5 Usage Scenarios

The purpose of this section is to provide a number of examples showing how to use Hourglass to measure different kinds of scheduling behavior. We are *not* attempting a good comparative evaluation of the different operating systems; for this reason and because the effects being shown are easily repeatable from run to run, we usually omit confidence intervals for the data being presented.

The scenarios in this section are intended to tell a story. We begin with two ways to measure context switch costs — these provide useful bits of knowledge but often have little direct impact on application programmers. The next example shows how to measure dispatch latency, which is a useful metric for real-time application developers but is still in the operating system domain. The fourth example moves into the application domain by directly measuring the ability of a single application to meet its deadlines while the operating system is busy doing other things; the example after that focuses on using a response time analysis to make predictions about the interaction between several real-time applications. Finally, the last example is about *CPU reservations*, a scheduling abstraction for guaranteeing that the requirements of multiple real-time applications will be met even when there is no coordination or cooperation between applications.

All performance data for this paper were taken on a uniprocessor 850 MHz Pentium III with 512 MB of RAM. Linux experiments were run on kernel version 2.4.17 unless otherwise specified. The high-resolution timer patch [1] is version 2.4.17-2.0. The preemptible kernel uses Robert Love's preempt-kernel-rml-2.4.17-3 and lock-break-rml-2.4.17-2 patches [12]. TimeSys Linux/GPL [23] is their version 3.0.108, based on Linux 2.4.7. FreeBSD measurements were taken on 4.5-RELEASE, and on that platform Hourglass uses the Linuxthreads package in order to get kernel threads. Windows 2000 measurements were taken on service pack 2.

5.1 Measuring the Direct Cost of Context Switches

One of the most straightforward uses of Hourglass's ability to measure gaps in thread execution is to infer the cost of running the kernel context switch code. To do this, invoke Hourglass with:

```
hourglass -d 60s -n 10 -a -p NORMAL -w CPU
```

or:

```
hourglass -d 60s -n 10 -a -p NORMAL \  
-w CPU_YIELD 0.9ms
```

These commands cause Hourglass to run for 60 seconds after creating ten threads, all running at the default time-sharing priority, all of which are CPU-bound. The only difference between the two commands is that the second causes each thread to yield the processor each time it has received 900 μ s of processor time, permitting us to distinguish between voluntary context switches (caused by yielding) and involuntary context switches (quantum expirations). Involuntary context switches are more expensive because they always occur at the same time as a clock interrupt. The output from these commands contains many lines of the following form:

```
0 442.323192 443.222287 0.899095 0.002184  
1 443.224485 444.123644 0.899159 0.002198  
9 444.125827 445.024925 0.899098 0.002183
```

The first column is the thread id, the second and third columns show the start and finish times of a gap-free interval of CPU time that the thread received, the fourth column is the duration of the interval (the difference between the second and third columns), and the last column is the "gap" between the start of the record being reported and the end of the previous record. All times are in milliseconds.

Hourglass is distributed with a Perl script `process_ctx` that produces a histogram of context switch times from Hourglass output, and also performs some simple statistical analyses. Figure 4 shows some histograms of context switch times for thread quantum expirations on several different OS configurations. These histograms reveal some interesting performance characteristics of the different systems. First, we can see that Linux with high-res timers appears to have a more efficient clock interrupt handler than standard Linux, while the TimeSys kernel appears to be a bit less efficient than the high-res kernel but is still more efficient than standard Linux. Second, we see that RML's preemptible kernel and lock breaking patches don't appear to affect context switch time. Third, the TimeSys Linux/GPL kernel, Windows 2000, and FreeBSD all cause about 50 context switches per second, rather than about 18 context switches per second for the Linux kernels based on 2.4.17. In other words, a modern stable Linux kernel provides CPU-bound threads with 60 ms quanta; TimeSys Linux (which is based on 2.4.7), Windows 2000, and FreeBSD give them 20 ms quanta; the kernel distributed with RedHat Linux 6.2 causes only about 7 context switches per second. The context switch cost for FreeBSD 4.5 kernel threads is over 30 μ s — considerably higher than the other OSs measured. This is because FreeBSD 4.5 Linuxthreads have not been optimized, and fail to take advantage of the opportunity to avoid page table operations when switching between

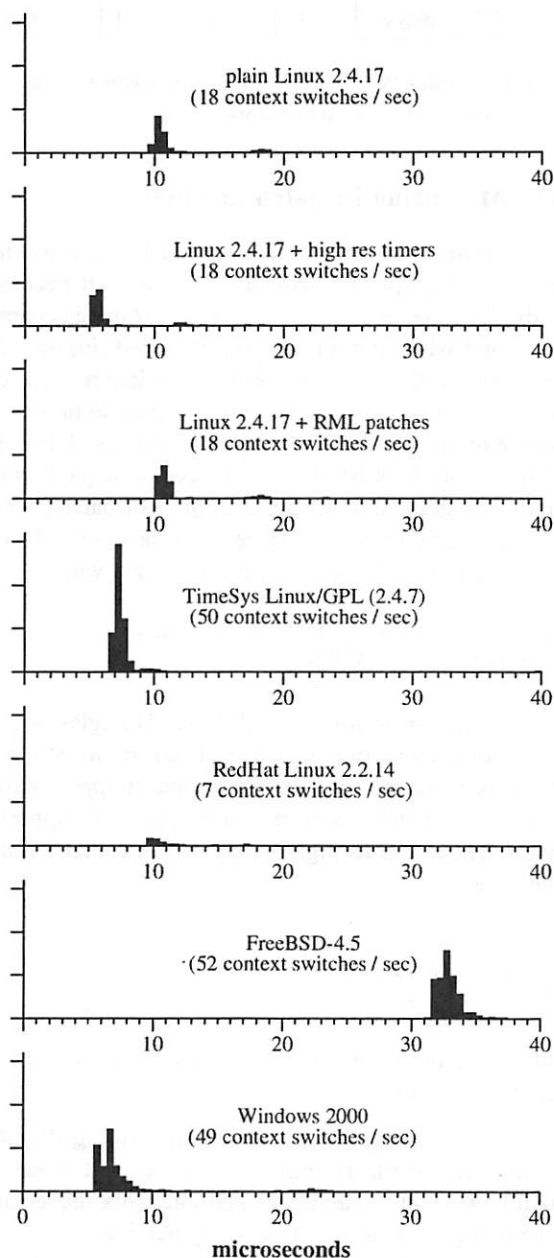


Figure 4: Histograms of times for involuntary context switches (e.g. caused by quantum expirations) between CPU-bound threads within a single process

threads in the same address space. We verified this by comparing the costs shown for FreeBSD in Figure 4 with the cost to context switch between processes. There was no statistically significant difference.

5.2 Measuring the Indirect Cost of Context Switches

Time spent in the kernel accounts for only one of the costs of context switches. Another cost, caused by lost cache locality, is potentially more severe. To see this, consider a thread whose working set in the L2 cache is 100 KB. If this thread is descheduled and, when scheduled again, starts running on a cold cache, it will spend time executing slowly until its working set again completely resides in the cache. The 850 MHz machine that we use has a memory read bandwidth of approximately 300 MB/s, so reading the 100 KB will take about $333 \mu\text{s}$ — far longer than the context switches that we saw in the previous section that took on the order of $10 \mu\text{s}$. This explains why scheduling quanta have not gotten much smaller over the past few decades, even though microprocessors have gotten thousands of times faster: for applications with non-trivial cache state, context switch times are tied to DRAM speeds, which grow much more slowly than processor speeds. For example, 1 ms scheduling quanta would lead to extremely good average application response time — this would be great for multimedia and interactive applications. However, instead of causing a 1% overhead, as the $10 \mu\text{s}$ context switch costs from the previous section might lead us to believe, this would create a 33% overhead if the average application has a 100 KB working set in the L2 cache.

Measuring application slowdown due to lost cache locality requires a more subtle technique than the one we used in the previous section: rather than measuring gaps, we must directly measure progress made by the application. The experimental methodology is as follows. To model a thread with, say, a 32 KB working set, Hourglass creates a thread that allocates 32 KB and repeatedly scans through it. For the control, Hourglass creates a single thread with a specified working set and measures how much “work” it can accomplish during a specific period of time. Each pass over the thread’s working set is considered to be a unit of work. We can then compare this amount of work with the total work performed by ten threads during the same time period. With all other factors being equal, any lost work for the ten-thread case must be caused by context switch overhead. The performance penalty C per context switch can be computed as

$$C = \left(\frac{W_1 - W_{10}}{W_1} \right) \left(\frac{T}{N_{10}} \right)$$

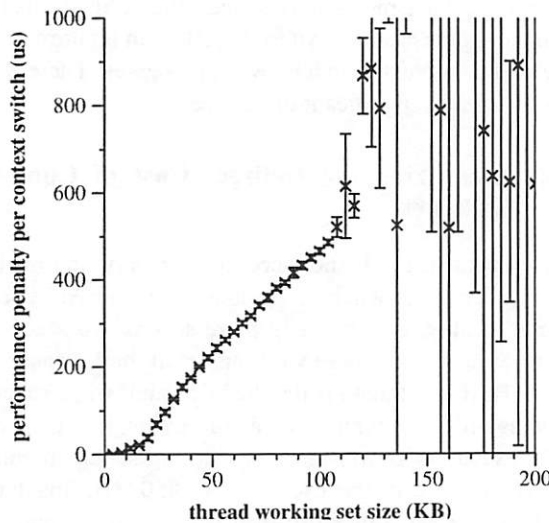


Figure 5: Thread working set size vs. performance lost to context switches on an 850 MHz Pentium III

where W_1 and W_{10} are respectively the total amount of work done by one and ten threads, T is the duration of an experimental run, and N_{10} is the number of context switches that occurred during the ten-thread run. In other words, the first term represents the fraction of work lost due to context switches; when multiplied by the duration of the experiment this gives the total performance penalty due to context switches; dividing by the number of context switches gives the penalty per switch.

Figure 5 shows the results of running this experiment for a variety of thread working set sizes on Linux 2.4.17. For each working set size, 15 trials were run; each trial consisted of a 45-second run of Hourglass with one thread, and a 45-second run with ten threads. Confidence intervals were calculated at 95% using the t -test [7, pp. 209–211]. The surprising thing about this graph is not how bad the results are (in fact, the results are roughly in line with the back-of-the-envelope calculation from the beginning of this section), but how unpredictable the data becomes for working set sizes over 128 KB (half the size of the L2 cache on a Pentium III Coppermine chip). We don't have a good explanation for this, but it probably has to do with the lack of page coloring logic in the Linux memory allocator — this can lead to cache conflict misses within a single application that are very unpredictable from run to run. Similar experiments, when run on FreeBSD, produced much more predictable results (see [5] for some discussion of the FreeBSD VM system, including page coloring optimizations).

Linux version	Samples later than:			
	1 ms	5 ms	10 ms	50 ms
2.2.14 RedHat	179	112	112	0
2.4.17 plain	284	56	45	1
2.4.17 RML	202	8	2	0
2.4.7 TimeSys	0	0	0	0

Table 1: Dispatch latencies for various Linux kernels with contention from filesystem and network activity

5.3 Measuring Dispatch Latency

A real-time operating system should be able to start running a high-priority thread shortly after it becomes ready. The time interval between when a thread becomes ready and when it begins to run is called *dispatch latency*. Dispatch latency is a serious problem for applications such as real-time audio where failure to produce a sample on time can result in audible artifacts. The point of the various lock-breaking and preemption patches for Linux is to get a substantial reduction in dispatch latency without overly increasing system overhead or reducing system maintainability. Running Hourglass with

```
hourglass -d 6m -n 1 -t 0 -p RTHIGH \
-w LAT 5.3ms -i RTC
```

creates a high-priority thread that uses Hourglass's built in real-time clock timers to wake itself up every 5.3 ms (the reason for the funny period is that it approximates a multiple of the power-of-two frequencies supported by the RTC). The Hourglass output then contains many lines like

```
latlate: 97.843206
latlate: 54.101933
latlate: 62.381242
```

indicating, in microseconds, the lateness of successive timer expirations.

Our experimental procedure for measuring dispatch latency was to run Hourglass in latency test mode for six minutes while running background work that creates scheduling contention. The background work that we chose was to concurrently (1) run a recursive `grep` over about 900 MB of data in about 60,000 files that is NFS3-mounted over a loopback connection, and (2) run a recursive `grep` over another copy of the same data that is NFS3-mounted with the server running on a separate machine. This is not necessarily a good choice for a real-world latency test, but it's fine for a demonstration.

Table 1 shows the results of this experiment for several Linux kernel versions where the total number of samples taken per operating system version is about

Linux version	Missed deadlines	Throughput
control 1	0.0%	—
control 2	—	94 Mbps
2.4.17 plain	33.0%	68 Mbps
2.4.17 RML	0.4%	66 Mbps
2.4.7 TimeSys	0.0%	59 Mbps

Table 2: A real-time thread that requires 4 ms of CPU time during every 5 ms can miss deadlines due to Ethernet receive processing

65,000. The results are pretty much what we would expect: the standard Linux kernels are fine most of the time, but they display occasional high latencies. The kernel with RML patches (supporting preemption and breaking of some locks [12]) does better, and the more aggressively modified TimeSys Linux/GPL kernel [23] never shows a latency above about 600 μ s.

5.4 Meeting Application Deadlines during Receive Processing

Although average- and worst-case dispatch latency are useful metrics for characterizing real-time operating systems, they only tell part of the story. As far as applications are concerned, what is important is not when they *start* running, but when they *finish*. In this section we show how kernel activity, in particular network receive processing, can affect applications' ability to get work done. Other kinds of receive processing, such as disk and USB, can also cause scheduling problems [17]. The experimental procedure was to hit the test machine with a full-speed TCP stream from another machine over 100 Mbps Ethernet using *netperf* at the same time that a real-time application attempts to meet its deadlines. If the receive processing interferes too much with application execution, deadlines will be missed.

The real-time application has very demanding requirements: it is a periodic thread that requires 4 ms of CPU time during each 5 ms period. This models a thread used for professional-quality real-time digital audio, where the end-to-end processing latency for sound data must be extremely low. The thread has the highest possible priority and uses real-time clock timers to schedule itself. The command line that does this is:

```
hourglass -d 20s -n 1 -t 0 -p RTHIGH \
-w PERIODIC 4ms 5ms -i RTC
```

We first ran two controls: one with only the real-time task, and one with only network receive processing. The results, shown in the first two lines of Table 2, were that all of the versions of Linux that we tested can meet

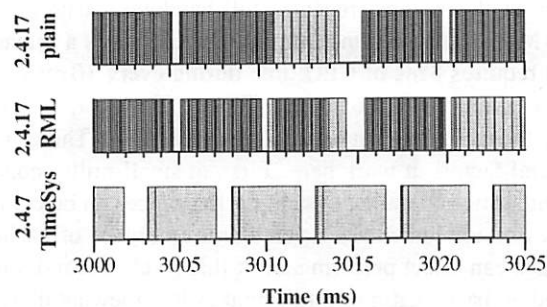


Figure 6: The effects of Ethernet receive processing on the execution of a periodic real-time thread

all deadlines on an otherwise quiet machine (control 1), and that a machine running only *netperf* can receive 94 Mbps of Ethernet data (control 2). The third line in Table 2 shows that plain Linux permits the application to miss about a third of its deadlines, but that it achieves the highest throughput. The preemptible Linux kernel with lock breaking patches misses less than one percent of its deadlines (53 out of about 12,000 deadlines, to be precise) and it gets only marginally lower throughput than plain Linux. Finally, the TimeSys Linux/GPL kernel misses no deadlines at all, but at the cost of reduced network throughput.

The tradeoff here is clear: by more strictly respecting the programmer's choice to run a user-level thread in preference to kernel activity, the two real-time-enhanced kernels permit real-time workloads to miss fewer deadlines. However, this causes network processing to be delayed, resulting in lower throughput. Figure 6 shows 25 ms segments of the execution traces generated by Hourglass while running these experiments. The top trace, generated on plain Linux, shows receive processing — interrupt and bottom-half handlers — interfering with thread execution to a significant extent. In the middle execution trace the 5 ms periodic structure of the real-time thread's execution is a little more apparent. However, there is still considerable interference: this is because RML's preemptible kernel patches only apply to the top half of the kernel; interrupts and bottom-half handlers can still freely run during the execution of a high-priority thread. Finally, the execution of the TimeSys Linux/GPL kernel shows that the real-time thread almost entirely locks out Ethernet processing: in this kernel both bottom-half handlers and most of each interrupt handler are run in thread context, and are therefore preemptible by user-mode threads.

The results of this experiment are sensitive to a number of parameters. For example, although plain Linux 2.4.17 did not permit a thread requiring 4 ms of CPU time during every 5 ms to meet most of its dead-

lines while the host receives full-bandwidth data over 100 Mbps Ethernet, under the same conditions a thread that requires 8 ms of CPU time during every 10 ms can meet essentially all of its deadlines. However, in this case the incoming bandwidth is only 52 Mbps. There are several factors at work here. First, at small-millisecond granularities operating system performance can be complex and unpredictable: even a small number of cache misses can affect performance at this level. Second, the window-based nature of TCP makes it somewhat undesirable for use in performance studies like this because its behavior may be highly nonlinear with respect to other factors in the experiment. The reduced bandwidth for the 10 ms periodic thread is most likely a result of the longer blocks of CPU time allocated to the real-time thread, which starves the `netperf` application for long enough that the sender must back off. Also, it is important to realize that 100 Mbps Ethernet puts a fairly modest load on a modern CPU: gigabit Ethernet, FireWire, and USB 2.0 can all potentially cause much larger receive processing loads than the ones measured here.

In summary, doing this kind of performance analysis “right” is very difficult because (1) the results are often highly sensitive to details of the how the experiment is performed, and (2) tuning the real-time performance of an OS kernel is inherently a matter of tradeoffs, and different people have different metrics for success. For example, it is possible to find otherwise reasonable people who are almost solely concerned with any one of the following metrics: average-case application throughput, real-time response, kernel code cleanliness and clarity, scalability to large multiprocessors, and scalability to small embedded devices.

5.5 Determining whether Concurrent Applications can meet Deadlines

The previous section addressed the ability of a single task to meet its deadlines under adverse conditions. In this section we consider multiple tasks. The motivation for this includes scenarios where, for example, a user is concurrently using a software modem, playing a game or listening to music, and encoding live video in the background. Modern PC hardware is capable of performing all of these tasks concurrently, provided that the operating system schedules each application at the proper times.

The workload is comprised of two tasks. The first task models either audio or software modem processing, and requires that a thread receive 3 ms of processor time during every 8 ms period. Software modems [10] and real-time audio can have tight latency requirements that are in this range. The second task represents a video

application that must maintain about 30 frames per second: it requires 17 ms of CPU time during each 33 ms period. We assign priorities to the two tasks in rate-monotonic fashion: the application with the shorter period gets a higher priority. Then, we use a standard real-time schedulability analysis to see if each application can be expected to meet its deadlines. The analysis returns answers as *worst-case response times* — the longest possible time between a task becoming ready and finishing its computation after taking into account all possible interleavings between the tasks [24]. If the response time of a task is less than or equal to its period, the task should work; otherwise it probably will not. In this example the response time of Task 0 is 3 ms, and Task 1 is 29 ms. In other words, there is no possible interleaving of the executions of the two tasks that can cause Task 0 to finish more than 3 ms after it becomes ready to run (this should be obvious — since Task 0 has higher priority, Task 1 cannot interfere with its execution), and Task 1 will always finish its work by 29 ms after it becomes ready, leaving 4 ms of slack before the end of its period.

To model this scenario in Hourglass, we run

```
hourglass -n 2 \
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \
-t 1 -p RTLOW -w PERIODIC 17ms 33ms -i NATIVE
```

Here, we give Task 0 a high-resolution timer to model the fact that we are considering it to be interrupt-driven, where the interrupts come from either the soft modem or audio hardware. Task 1, on the other hand, uses `usleep()` to effect wakeups. The result of this experiment on a Linux kernel with the high-resolution timer patch contains lines that look something like

```
thread 0: missed 0 deadlines, hit 1001
thread 1: missed 142 deadlines, hit 143
```

meaning that Task 1 misses about as many deadlines as it hits. There are two reasons for the disagreement between the response time analysis and Hourglass output. First, the 10 ms granularity of `usleep()`-based timers results in *release jitter*. Release jitter occurs when a thread is conceptually ready to run earlier than the operating system becomes ready to run it. To see how this happens here, consider the case where Task 1, the video application, finishes a frame just four or five milliseconds before it is due to start processing the next frame. It then puts itself to sleep, but may have to wait 10 ms for the next clock interrupt to arrive and wake it up. The added delay can cause it to miss deadlines. The second reason for disagreement is that if a thread in the real-time priority class calls the Linux implementation

of `usleep()` with an argument of 2 ms or smaller, the kernel will actually busy-wait until the expiration time. This has the effect of increasing the worst-case run time of Task 1 from 17 ms to 19 ms.

Fortunately, the response time analysis is able to take release jitter into account. Re-running the analysis for Task 1 with 8 ms of release jitter and a 19 ms worst-case compute time shows that its worst-case response time is 39 ms. This is longer than its period, and therefore Task 1 is not guaranteed to meet all deadlines.

Since the task set is *infeasible*, or not guaranteed to allow all tasks to meet all deadlines, we now consider ways to make it feasible. One way is to schedule Task 1's wakeups using a timer that causes less release jitter and avoids the harmful busy-waiting; another way is to decrease the run-time of one of the tasks enough that the tasks become feasible in spite of these factors. To test the first method, we invoke Hourglass with

```
hourglass -n 2 \  
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \  
-t 1 -p RTLOW -w PERIODIC 17ms 33ms -i HR
```

which produces these results

```
thread 0: missed 0 deadlines, hit 1003  
thread 1: missed 0 deadlines, hit 304
```

on a Linux kernel with the high-res timers patch. This corroborates the initial response time analysis in this section that predicted that the task set is feasible.

To apply the second method, assume that we have reduced the execution time of Task 1 to 12 ms by decreasing the resolution of the video display or by making use of a hardware accelerator. The response time analysis indicates that in this case all deadlines can be met. We test this by running

```
hourglass -n 2 \  
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \  
-t 1 -p RTLOW -w PERIODIC 12ms 33ms -i NATIVE
```

which gives these results:

```
thread 0: missed 0 deadlines, hit 1001  
thread 1: missed 0 deadlines, hit 303
```

Papers by Audsley et al. [3] and Tindell et al. [24] are good sources of information about response time analysis for real-time task sets. SPAK [15] is a static priority analysis kit that implements response time analyses for a variety of task set assumptions, as well as a task simulator that can serve as a middle ground between response-time analysis and Hourglass for testing the ability of task sets to meet their deadlines.

5.6 Using CPU Reservations

The response time analysis used in the previous section requires a *closed system*, where all real-time applications and their requirements must be known before the analysis can be performed. However, when real-time applications are run on general-purpose operating systems like Linux the *open system* model is usually more appropriate: real-time applications can begin or end at any time, and applications can change their requirements freely. The standard Linux kernel can be used as an open system, but there is a problem: without knowing the requirements of the other applications, it is impossible to know, for a given application, what priority should be assigned to it in order to allow it to operate correctly, and to not interfere with existing applications. This lack of coordination between applications can lead to *priority inflation* where developers overestimate the priority at which their application should run, since a badly performing application reflects negatively on its developer. In the extreme case, this results in excessive migration of code into interrupt handlers [10]. Also, priority-based scheduling suffers from the problem that a misbehaving application can stop the progress of all applications running at lower priorities. Furthermore, rate-monotonic scheduling assigns priorities solely based on the periods of applications; this can result in anomalies where, for example, a critical application is given lower priority than an unimportant application.

In recent years scheduling solutions have been developed that address all of these problems. They go by different names, but the core abstraction is always some form of *CPU reservation*. For example, a video application could reserve 5 ms of CPU time during every 30 ms period, in order to ensure that it can always display 33 frames per second. If the operating system agrees to provide the reservation, it is essentially guaranteeing that the CPU time will always be available. This sort of guarantee relieves developers of the burden of finding a good priority for their application. However, it adds the non-trivial difficulty of specifying application requirements — this is probably the main reason that reservation schedulers have not yet made it into mainstream operating systems.

Hourglass provides direct support for making CPU reservations. Currently it supports only Linux/RK from CMU [11] and TimeSys Linux/CPU [23] (a proprietary extension to TimeSys Linux/GPL). However, reservation functionality is modular and support for new schedulers can be added with minimal effort.

To illustrate the use of CPU reservations, we return to the task set from the previous section. To run the

tasks in CPU reservations rather than using priority-based scheduling, we invoke Hourglass like this:

```
hourglass -n 2 \  
-t 0 -w PERIODIC 3ms 8ms -rh 3ms 8ms \  
-t 1 -w PERIODIC 17ms 33ms -rh 17ms 33ms
```

The resulting output contains the lines:

```
thread 0: missed 985 deadlines, hit 12  
thread 1: missed 106 deadlines, hit 197
```

Obviously things did not go well during this run. The mistake was to reserve exactly as much CPU time as each thread needed — this makes them vulnerable to minute perturbations in the schedule. Rather, we should reserve just a little extra CPU time like this:

```
hourglass -n 2 \  
-t 0 -w PERIODIC 3ms 8ms -rh 3.1ms 8ms \  
-t 1 -w PERIODIC 17ms 33ms -rh 17.1ms 33ms
```

The resulting output contains the lines:

```
thread 0: missed 0 deadlines, hit 1000  
thread 1: missed 0 deadlines, hit 303
```

For real applications, we would have done two things differently. First, we would have given them *soft* CPU reservations instead of *hard* CPU reservations — hard reservations enforce a strict upper limit on the amount of time a thread will receive, while soft reservations guarantee that applications will receive a minimum amount of CPU time, but permit them to receive extra time if it is available. Hourglass accepts the `-rs` command-line option to provide a soft reservation. Second, rather than just reserving 0.1 ms of extra CPU time per thread per period, we would tune the amount of over-reservation to match the application's importance (e.g. a critical application gets a bigger reservation, all other things being equal) and its expected variability (e.g. an application with high variance in run-time, such as an MPEG video decoder, would get a larger reservation than a more predictable application).

6 Related Work

Hourglass fits into the general methodology that has been called *gray-box systems* [2]. The name refers to the fact that these techniques make inferences about system behavior by combining the results of observations with general knowledge about internal structure — the operating system is treated as a mixture between a black-box and a white-box.

Section 2 compared Hourglass with other ways to measure or infer scheduling behavior. As far as we know, Hourglass has no direct competition — there are no publicly available tools that occupy the same niche. However, Hourglass was influenced by `txofy`, a tool developed by Mike Jones and others for internal use at Microsoft Research; it was instrumental in producing data for a number of papers [8, 9, 16, 17, 18]. The gap detection algorithm from Section 3.1 was first developed for `txofy`. However, the two tools have diverged considerably: `txofy` was a one-off tool specifically designed to monitor and debug CPU reservation schedulers; it supports only a single source of timers, a single thread model, and a static set of threads (it has, however, been modified to support multiprocessors). Hourglass, on the other hand, is a general framework for making inferences about scheduling behavior; it is portable and extensible, and provides direct support for all techniques described in Section 5.

7 Availability and Status

Hourglass was written from scratch and is released under a BSD-style license. It runs on Linux, FreeBSD, and Windows 2000. The Hourglass home page is:

<http://www.cs.utah.edu/~regehr/hourglass>

Since web links tend to die, in the long run we suggest the alternate strategies of looking for Hourglass at the Flux Group web site: <http://www.cs.utah.edu/flux> or using the dominant web search engine to look for “regehr” and “hourglass.”

Hourglass currently supports uniprocessor x86-based systems. Both of these limitations could be addressed in straightforward ways. For example, Alphas support the `rpcc` instruction and UltraSPARCs have a `%tick` register; these are basically equivalent to the `rdtsc` cycle counter on x86 machines. To support multiprocessors Hourglass would require that the cycle counters on all processors are synchronized. Although Hourglass could not tell which processor each thread runs on without help from the kernel, it is possible to infer a consistent timeline from a multiprocessor execution trace. For example, if it is observed that Thread 1 receives a continuous segment of CPU time while Thread 2 stops running and Thread 3 starts, we can infer that Thread 1 was on one processor, and on a different processor Thread 2 was preempted in order to run Thread 3.

A more brute-force approach to generating accurate execution traces on multiprocessors would involve writing a minimal kernel extension that overloads the high-order bits of the cycle counters in such a way that any timestamp value could be identified as coming from a

particular processor — kernel support is required because writing to the model-specific x86 registers is a privileged operation. This approach would give accurate, unambiguous multiprocessor execution traces but would cause problems if the kernel or some other application relies on accurate or synchronized timestamp counters.

Machines such as laptops and Pentium 4s can vary their clock speeds dynamically to deal with heat and power issues, making `rdtsc` less useful or useless. We have not attempted to address this problem since at present it's easy to avoid these processors. Dynamic speed scaling is not only a problem for programs like Hourglass, but also for actual real-time applications that depend on getting a certain amount of work done during a given time period.

Finally, an inconvenience of Hourglass on Unix-like systems is that users need root privileges to create threads in the real-time priority class and to pin pages into physical memory.

8 Conclusion

The contributions of this paper have been (1) to show that a lot of detail about operating system scheduling behavior can be inferred using an instrumented application and a little knowledge about internals, (2) to present some new gray-box inference techniques for learning about scheduling behavior, and (3) detailed descriptions of how to use a freely available tool to perform these measurements.

Although there are tools for measuring scheduling latency [19], Hourglass is a more comprehensive user-level real-time test application than any other software that we know of. It has been invaluable for discovering what is really happening to applications at the granularity of microseconds and milliseconds. We believe that this kind of application will become increasingly important as multimedia and other soft real-time applications become a more important part of the mix of programs that users run.

Acknowledgments

The author would like to thank Jason Baker, Paul Davis, Toon Moene, Mac Newbold, and Pat Tullmann for providing constructive feedback on drafts of this paper. Eric Eide provided graph and LaTeX expertise. Terry Lambert and others on the FreeBSD-hackers mailing list helped with analysis of performance on that OS. The Utah Emulab folks provided a great platform for running the experiments reported here.

This work was supported, in part, by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreements F30602-99-1-0503 and F33615-00-C-1696.

References

- [1] George Anzinger. The Linux High Resolution Timers Project. <http://high-res-timers.sourceforge.net>.
- [2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, November 2001.
- [3] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
- [4] Stephen Childs and David Ingram. The Linux-SRT integrated multimedia system: Bringing QoS to the desktop. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.
- [5] Matthew Dillon. Design elements of the FreeBSD VM system. *Daemon News*, January 2000. http://www.daemonnews.org/200001/freebsd_vm.html.
- [6] Ashvin Goel and Jonathan Walpole. Gscope: A visualization tool for time-sensitive software. In *Proc. of the 2002 USENIX Annual Technical Conf. FREENIX Track*, Monterey, CA, June 2002.
- [7] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.
- [8] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation experience on Windows NT. In *Proc. of the 3rd USENIX Windows NT Symposium*, pages 93–102, Seattle, WA, July 1999.
- [9] Michael B. Jones, John Regehr, and Stefan Saroiu. Two case studies in predictable application scheduling using Rialto/NT. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 157–164, Taipei, Taiwan, May 2001.
- [10] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.
- [11] Linux/RK: A Resource Kernel for Linux. <http://www-2.cs.cmu.edu/~rajkumar/linux-rk.html>.
- [12] Robert Love. Preemptible kernel and lock-breaking patches for Linux. <http://www.tech9.net/rml/linux>.
- [13] Andrew Morton. Low Latency Linux. <http://www.zip.com.au/~akpm/linux/schedlat.html>.

- [14] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernels: A resource-centric approach to real-time and multimedia systems. In *Proc. of the SPIE/ACM Conf. on Multimedia Computing and Networking*, pages 150–164, San Jose, CA, January 1998.
- [15] John Regehr. SPAK: a static priority analysis kit. <http://www.cs.utah.edu/~regehr/spak>.
- [16] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001. <http://www.cs.utah.edu/~regehr/papers/diss>.
- [17] John Regehr and John A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 141–148, Taipei, Taiwan, May 2001.
- [18] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, December 2001.
- [19] Benno Senoner. Latencytest. <http://www.gardena.net/benno/linux/audio>.
- [20] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.
- [21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, June 1994.
- [22] Vijay Sundaram, Abhishek Chandra, Pawan Goyal, Prashant Shenoy, Jasleen Sahni, and Harrick Vin. Application performance in the QLinux multimedia operating system. In *Proc. of the 8th ACM Conf. on Multimedia*, Los Angeles, CA, November 2000.
- [23] TimeSys Linux/GPL. <http://www.timesys.com/products>.
- [24] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(2):133–151, March 1994.
- [25] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 246–255, Phoenix, AZ, December 1999.
- [26] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proc. of the USENIX 2000 Annual Technical Conf.*, pages 1–14, San Diego, CA, June 2000.

A Command Line Arguments

This is Hourglass 0.5

usage: hourglass <options>

global options:

- n <number of threads to create> -- this is the only mandatory option
- d <duration of experiment> (default is 10s)
- c -- print raw execution trace (in cycles and not based at zero) in addition to standard trace (which starts at zero and is in ms)
- t <thread number for subsequent options to affect in the range 0..n-1>
- a -- subsequent per-thread options apply to all threads; this remains in effect until the next '-t' option is encountered
- e -- number of records in execution trace; default is 300000 (about 5 MB)

per-thread options:

- p <priority> -- priority is one of: IDLE, LOW, NORMAL, HIGH, HIGHEST (timesharing -- default is NORMAL) RTLOW, RTMED, RTHIGH (real-time)
- rh <amount> <period> -- request a hard CPU reservation
- rs <amount> <period> -- request a soft CPU reservation
- w <workload>
 - CPU -- cpu-bound execution trace (default)
 - CPU_SCAN <size> -- cpu-bound and also scans an array that is size KB long
 - CPU_YIELD <amount> -- yield the processor after getting <amount> of CPU time
 - CPU_SCAN_YIELD <size> <amount> -- combine scanning and yielding behaviors
 - PERIODIC <amount> <period> -- periodic task, make execution trace
 - CPU_PERIODIC <amount> <period> -- cpu-bound periodic (see documentation)
 - LAT <period> -- dispatch latency test
- i <timer>
 - NATIVE -- native Unix or Windows timers (default)
 - HR -- high resolution timers (if available: requires a patched kernel)
 - RTC -- timers based on Linux real-time clock (if available)
 - MM -- multimedia timers (Win32 only)
- s <time> -- start running this thread <time> units into the Hourglass run
- f <time> -- terminate this thread <time> units into the Hourglass run

Times are self-describing; valid examples are: 3m, 1.5s, 1020ms, 87.0us.

A Decoupled Architecture for Application-Specific File Prefetching

Chuan-Kai Yang Tulika Mitra Tzi-Cker Chiueh

Computer Science Department

Stony Brook University

Stony Brook, NY 11794-4400

ckyang@cs.sunysb.edu, tulika@comp.nus.edu.sg, chiueh@cs.sunysb.edu

Abstract

Data-intensive applications such as multimedia and data mining programs may exhibit sophisticated access patterns that are difficult to predict from past reference history and are different from one application to another. This paper presents the design, implementation, and evaluation of an automatic application-specific file prefetching (AASFP) mechanism that is designed to improve the disk I/O performance of application programs with such complicated access patterns. The key idea of AASFP is to convert an application into two threads: a *computation* thread, which is the original program containing both computation and disk I/O, and a *prefetch* thread, which contains all the instructions in the original program that are related to disk accesses. At run time, the prefetch thread is scheduled to run sufficiently far ahead of the computation thread, so that disk blocks can be prefetched and put in the file buffer cache before the computation thread needs them. Through a source-to-source translator, the conversion of a given application into two such threads is made completely automatic. Measurements on an initial AASFP prototype under Linux show that it provides as much as 54% overall performance improvement for a volume visualization application.

1 Introduction

With the emergence of data-intensive applications such as multimedia and data mining workloads, disk I/O performance plays an ever more critical role in the overall application performance. This is due to the increasing performance gap between CPU speed

and disk access latency. To improve performance for these data-intensive applications requires that disk access delays be effectively masked or overlapped with computation. Many application programmers alleviate this performance gap problem by manually interleaving computation with synchronous disk I/O. That is, the program issues a disk I/O call, waits for the I/O to complete, performs some computation, then issues another disk I/O request, and so on. Another solution to mask disk I/O delay is asynchronous I/O. Although conceptually straightforward, asynchronous disk I/O tends to complicate program logic and possibly lengthen software development time. Also, an asynchronous disk I/O-based application written for one disk I/O subsystem may not work well for another with different latency and bandwidth characteristics. Yet another solution is to cache recently accessed disk blocks in main memory for future reuse. If there is a high degree of data reuse, file or disk caching can reduce both read latency and disk bandwidth requirements. However, for many media applications caching is not effective either because the working set is larger than the file buffer cache, or because each disk block is used only once.

A well-known solution to this problem implemented in many UNIX systems, including Linux, is to prefetch a disk block before it is needed [12]. Linux assumes that most applications access a file in a sequential order. Hence, whenever an application reads a data block i from the disk, the file system prefetches blocks $i + 1$, $i + 2$, ... $i + n$ for some small value of n . If the access pattern is not sequential, prefetching is suspended until the application's disk accesses exhibit a sequential access pattern again. For most common applications, this simple sequential prefetching scheme seems to work well. However, sequential access is not necessarily the dominant access pattern for some other impor-

tant applications, such as volumetric data visualization, multi-dimensional FFT, or digital video playbacks (e.g., fast forwards and rewinds); these are popular applications in the scientific visualization or multimedia world.

Different applications may exhibit different access patterns and thus a fixed prefetching policy is not adequate. We propose an *Automatic Application-Specific File Prefetching* mechanism (AASFP) that allows an application program to instruct the file system how to prefetch on its behalf. AASFP automatically generates the prefetch instructions from an application's source code. The prefetch instructions are instantiated as a program running inside either a local file system or a network file server. AASFP is an application of the idea of *decoupled architecture* [22], which was originally proposed to address the von Neumann bottleneck to bridging the CPU-disk performance gap.

AASFP transforms a given application program into two threads: a *computation thread* and a *prefetch thread*. The computation thread is the original program and contains both computational and disk access instructions; the prefetch thread contains all the instructions in the original program that are related to disk I/O. At run time, the prefetch thread is started earlier than the computation thread. Because the computation load in the prefetch thread is typically a small percentage of that of the original program, the prefetch thread could remain ahead of the computation thread throughout the application's entire lifetime. Consequently, most disk I/O accesses in the computation thread are serviced directly from the file system buffer, which is populated beforehand by the I/O accesses in the prefetch thread. In other words, the prefetch thread brings in exactly the data blocks as required by the computation thread before they are needed. Of course, it is not always possible for the prefetch thread to maintain sufficient advance over the computation thread. For example, the computation thread may need to access a disk address, but the generation of this address may depend on the user inputs or on the inputs from a file. In such cases, these two threads need to synchronize with each other. Fortunately, such tight synchronization is relatively infrequent in I/O-intensive programs. The key advantage of AASFP is that it eliminates the need for manual programming of file prefetch hints by generating the prefetch thread from the original program using compile-time analysis. In addition to being more accurate in *what* to prefetch, AASFP also pro-

vides more flexibility to the file system in deciding *when* to prefetch disk blocks via a large look-ahead window lead by the prefetch thread.

The rest of this paper is organized as follows. Section 2 reviews some of the related work. Section 3 describes the system architecture of AASFP. Section 4 shows how to generate the prefetch thread from a given program. Section 5 discusses the run-time system of AASFP. Section 6 evaluates the performance results of AASFP. Section 7 concludes this paper and points out potential future directions.

2 Related Work

Prefetching and caching have long been implemented in modern file systems and servers. However, until recently, prefetching was restricted to sequential lookahead and caching was mostly based on the LRU replacement policy. Unfortunately, sequential access is not necessarily the dominant access pattern for certain important data-intensive applications, such as volume visualization applications [27] and video playbacks involving fast forwards and rewinds. This observation has spawned research in three different directions.

Predictive Prefetching: Early file prefetching systems [6, 9, 10, 26] attempted to deduce future access patterns from past reference history and performs file prefetching based on these inferred access patterns. This approach is completely transparent to application programmers. However, incorrect prediction might result in unnecessary disk accesses and poor cache performance due to the interference between current working sets and predicted future working sets. Some different prediction policy is also possible. For example, the work by Duchamp et al. [7], in the context of Web page browsing, proposed to prefetch data based on the popularity of a Web page.

Application Controlled Caching and Prefetching: Patterson et al. [15, 17] modified two UNIX applications, *make* and *grep*, to provide hints to the file system regarding the files that applications are going to touch. The hints are given to the file system by forking off another process that just accesses all the files that are going to be accessed by the original process ahead of time. This study showed a 13% to 30% reduction

in execution time. In a later paper, they modified a volume visualization application to provide hints and obtained reduction in execution time by factor of 1.9 to 3.7 [18].

While Patterson et al. were performing application-controlled prefetching, Cao et al. [2, 3] were investigating the effects of application hints on file system caching. Traditionally, the file system cache implements the LRU replacement policy, which is not adequate for all applications. Cao et al. proposed a two-level cache management scheme in which the kernel allocates physical pages to individual applications (*allocation*), and each application is responsible for deciding how to use its physical pages (*replacement*). This two level strategy, along with a kernel allocation policy of LRU with Swapping and Placeholders (LRU-SP), reduced disk block I/O by up to 80% and execution time by up to 45% for different applications.

Prefetching and caching are complimentary to each other. Cao et al. [4] first proposed an integrated prefetching and caching scheme which showed an execution time reduction of up to 50% through a trace-driven simulation study. Patterson et al. [19] showed another prefetching and caching scheme based on user-supplied hints which classifies buffers into three types: prefetching hinted blocks, caching hinted blocks for reuse, and caching used blocks for non-hinted access. Later, they extended their scheme to support multiple processes where the kernel allocates resources among multiple competing processes [25]. Joint work by both groups later on arrived at an adaptive approach that incorporates the best of both schemes [8, 24].

Tait et al. [23] adopted the idea of file prefetching, hoarding, and caching in the context of mobile computing. Rochberg et al. [21] implemented a network file system extension of the Informed prefetching scheme [16] that uses modified NFS clients to disclose the application's future disk accesses. The result showed a 17–69% reduction in execution time and demonstrated the scheme can be extended to network file systems quite easily. Pai et al. [14] applied the prefetching idea to manually construct the decoupled architecture where the computation performed by the Web server never blocks on I/O. Similarly, it was also adopted by the Unix backup (/sbin/dump) program (originally from Caltech), where the main dump program should never block on I/O.

Compiler Directed I/O: Instead of requiring application programmers to enter the disk access hints, Mowry et al. [13] described a compiler technique to analyze a given program and to issue prefetch requests to an operating system that supports prefetching and caching. This fully automatic scheme significantly reduces the burden on the application programmer and thus enhances the feasibility of application-specific prefetching and caching. However, this approach is restricted to loop-like constructs, where the disk access addresses usually follow a regular pattern. Recently, Chang and Gibson [5] developed a binary modification tool called SpecHint that transforms Digital UNIX application binaries to perform speculative execution and issue disk access hints. They showed that this technique can perform quite close to Informed Prefetching without any manual programming efforts. Chang's work is most similar to AASFP but is different in two ways:

- AASFP operates on the source code of application programs and constructs application-specific prefetch instructions as a separate run-ahead thread. Therefore, AASFP's lookahead window could be arbitrarily large, subject only to the synchronization constraint between the computation and prefetch threads. In contrast, Chang's approach is more limited because the extent of prefetching depends on the amount of CPU cycle time it can get scheduled at run time.
- AASFP's prefetch thread only executes disk I/O-related code, whereas SpecHint executes the original application speculatively to identify future disk access patterns. In addition, AASFP ensures that the prefetch thread always run sufficiently far ahead of the computation thread, whereas SpecHint runs the "pre-execution" pass only when the CPU is idle because of disk I/O.

3 System Architecture

The AASFP prototype consists of three components: a *source-to-source translator*, a *run-time prefetch library*, and a modified Linux kernel, as shown in Figure 1.

The source-to-source translator generates a prefetch

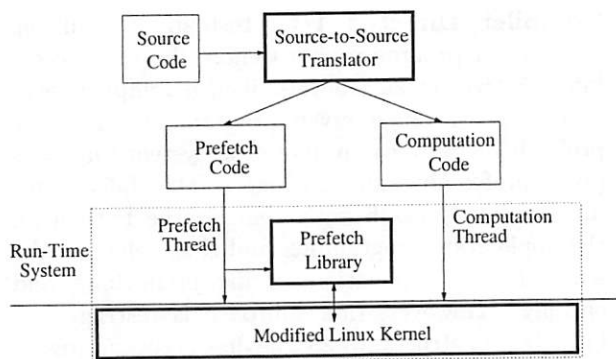


Figure 1: Software Architecture of AASFP

thread from an application program by extracting the parts of the program that are related to disk I/O accesses and removing all the other computation. In addition, all the disk I/O calls in the prefetch thread are replaced by their corresponding prefetch calls to the prefetch library. The original program itself forms the computation thread. There is a one-to-one correspondence between the prefetch calls in the prefetch thread and the actual disk I/O calls in the computation thread. The prefetch thread is executed as a Linux thread, as supported by the *pthread* library [11]. All the prefetch calls are serviced by AASFP's run-time prefetch library, which generates the logical file block address associated with each prefetch call and inserts the prefetch requests into a *user-level prefetch queue*. When the user-level prefetch queue becomes full, the prefetch library makes a system call to transfer the prefetch requests to the application's kernel-level prefetch queue. However, these prefetch hints are completely non-binding (i.e., the kernel might ignore these hints if there are not enough resources for file prefetching).

A major innovation of AASFP is that the computation and the prefetch thread are automatically synchronized so that the kernel neither prefetches too far ahead nor falls significantly behind. That is, AASFP ensures that the portion of the target files prefetched are those that are to be accessed by the computation thread in the immediate future. To maintain synchronization between the prefetch and computation threads, AASFP marks each entry in the prefetch queue with the ID of the corresponding prefetch call. The kernel also maintains the ID of the current disk I/O call of the computation thread. These IDs are created and maintained by the run-time system without programmer intervention. When the ID of an entry in the

prefetch queue is smaller than the ID of the most recently made disk I/O call, the computation thread has run ahead of the prefetch thread, and the kernel simply skips the expired prefetch queue entries. Therefore, even if the prefetch thread falls behind, it never prefetches unnecessary data. To prevent the prefetch thread from running too far ahead of the computation thread, the kernel attempts to maintain a prefetch depth of N , based on the following: the run-time measurements of the average disk service time, which can be measured off-line beforehand; and the average amount of computation that the application performs between consecutive I/O calls.

The files that an application accesses can be classified into two categories: *data files*, which are expected to be prefetched by AASFP, and *configuration files*, which are retrieved in the beginning and used to steer the computation to generate access addresses, and thus do not need to be prefetched. It is not possible for the translator to distinguish between data and configuration files. For every data file, the application programmer can optionally provide an additional annotation of the form *PREFETCH file *fp* to indicate to the source-to-source translator that the file descriptor *fp* should be prefetched at run time. These files will be referred to as *prefetchable files* for the rest of the paper.

3.1 Prefetch Thread

In this subsection we describe the basic operations in the prefetch thread and its interaction with the computation thread and the prefetch library.

Figure 2 shows how AASFP's translator converts an example application into a computation and a prefetch thread. The programming interface provided by the prefetch library includes the following four calls:

1. **create_prefetch_thread**
(**prefetch_function**): This function allows an application to fork a prefetch thread, and to execute the **prefetch_function**. The **prefetch_function** is passed as the input argument, as shown in line 1 of Figure 2.
2. **prefetch_XXX()**: These are a set of functions that the prefetch thread can use to specify prefetch calls. The prefetch calls re-

```

int fp;

COMPUTATION THREAD

void main(void){
    int i; int data[100];
    /* create the prefetch thread */
## C1. create_prefetch_thread(
        (void *)prefetch_function);
    /* open data file */
    C2. fp = open("mydata.dat", O_RDONLY);
    /* signal the prefetch thread */
## C3. synchronize(1,signal);

    C4. for (i=99; i>=0; i--){
        /* I/O */
    C5.     lseek(fp, i*4, SEEK_SET);
    C6.     read(fp, &data[99-i], 4);
        /* Computation */
    C7.     data[99-i] = data[99-i]*i;
    }
    /* close data file */
    C8. close(fp);
}

PREFETCH THREAD

void prefetch_function(void){
    int i;

    /* wait for file to open */
    $$ P1. synchronize (1,wait);
    /* inform prefetch library */
    $$ P2. inform_open(fp);
    /* prefetch */
    P3. for (i=99; i>=0; i--){
        /* only the I/O part is retained */
    P4.     prefetch_lseek(fp, i*4, SEEK_SET);
    P5.     prefetch_read(fp, 4);
    }
    /* inform prefetch library */
    $$ P6. inform_close(fp);
}

```

Figure 2: Computation thread (left) and prefetch thread (right). The main function without the lines marked by ## is the original application code. This code is modified to add a prefetch thread represented by the prefetch function. The lines in the prefetch function that are not marked with \$\$ are extracted from the original main function. Other additional functions are added for the two threads to communicate with each other. Notice that the computation part of the main function does not appear in the prefetch function

place the original I/O calls and use almost the same syntax. For example, for the I/O call `read(stream, ptr, size)`, AASFP provides the prefetch call `prefetch_read(stream, size)`. The parameter `ptr` for the data is not needed, since the prefetch call only generates the target data's starting address but never actually performs the real I/O. Lines P4 and P5 in Figure 2 represent the prefetch calls corresponding to the I/O calls at lines C5 and C6 of the main function respectively.

3. **inform_open(file_pointer), inform_close(file_pointer):** These two functions are used by the prefetch thread to inform the prefetch library that some file is opened or closed. Such notification is necessary so that the prefetch library can maintain a file table consisting of {file pointer, current offset} for those files accessed by the prefetch thread. Lines P2 and P6 in Figure 2 represent the inform calls corresponding to the *open* and *close* system calls in the computation thread.

4. **synchronize(synchronization_point, type):** This function synchronizes the two threads. The argument `type` can be `signal` or `wait`. Synchronization is discussed in detail in the next subsection.

3.2 Synchronization

The prefetch thread and the computation thread execute independently of each other until either of them reaches a synchronization point. Each of the following cases represents a synchronization point:

- **File Open:** The prefetch thread needs to wait until the computation thread opens the file. The two threads are synchronized by calling the `synchronize` function with an identical `synchronization_point` argument. The computation thread opens a file and calls the `synchronize` function to signal that the file has been opened. The prefetch thread calls

the `synchronize` function, and waits until it receives the signal. Lines C3 and P1 in Figure 2 represent the synchronization points for the two threads. The `synchronize` function is implemented using the `pthread_cond_wait` and `pthread_cond_signal` primitives of the `pthread` library. This synchronization is necessary because there may exist a conditional branch in the original code as to which data file will be used in the application. Having a synchronization point like this helps avoid prefetching a wrong file that will never be used in the future.

- *User Input:* The prefetch thread needs to wait for the computation thread if the address generation computation depends on some input from the user (`stdin`) or from some user-specified file. Only one of the threads is allowed to actually perform an I/O operation and hence the other thread needs to wait. If disk access address generation is dependent on data from a file that is being prefetched, AASFP puts synchronization points in both threads. This way, the prefetch thread can proceed only after the computation thread actually reads in the data from the file system buffer cache. For terminal input and input from files that are not prefetched (typically setup or configuration files), AASFP allows the prefetch thread to perform the terminal/disk I/O instead and removes those I/O requests from the computation thread. Thus, the prefetch thread can still stay ahead of the computation thread. In this case, the computation thread waits until the prefetch thread completes its I/O, and then the computation thread resumes execution.

- *Read after Write:* If a program involves only reading from one file and writing to a different file, or non-overlapping reads/writes on the same file, then there is no need for synchronization; otherwise, a synchronization is needed. Assuming the prefetch thread is running ahead and it detects a read request that is dependent on some previous write operation, it then stops and waits for the computation thread to finish the dependent write operation. Only after the associated write is done, regardless of whether it is a synchronous or an asynchronous write, can the prefetch thread proceed.

3.3 Data Sharing

Although the prefetch thread and the computation thread can share data through global variables in the application program, sometimes we may need to share information in other ways. AASFP provides the abstraction of a *communication channel* between the two threads. This communication channel is provided by the prefetch library and is implemented using a shared buffer between the two threads. Typically the only variables that need to be shared are the common file pointers and the user/file input variables. The following functions support sharing data between the two threads:

- **send_fileptr(file_pointer), receive_fileptr(&file_pointer):** These functions send and receive file pointers between the threads. The functions provide implicit synchronization and thus eliminate the need to call synchronization functions explicitly. Lines C3 and P1 in Figure 3 are examples.
- **send_XXX(), receive_XXX():** These are a set of functions that a thread uses to send/receive data to/from the other thread. They replace the I/O calls of the same name and use almost the same syntax. For example, for the I/O call `fscanf(fp, "%d", &value)`, AASFP provides the send call `send_fscanf(fp, "%d", &value)`, and the receive call `receive_fscanf("%d", &value)`. The send function reads in the value as well as sends it to the communication channel for the other thread to pick up. The variable `fp` is not needed in the receive call since the receive function receives it from the communication channel. As discussed in Section 3, the prefetch thread performs all the I/O operations on the configuration file; these operations are removed from the computation thread. In Figure 3, lines marked by XX are removed from the computation thread, and lines in the prefetch thread that are not marked by \$\$ are extracted from the original main function. Instead of `fscanf`, the prefetch thread performs a `send_fscanf`, and the computation thread performs a `receive_fscanf`. These functions also provide implicit synchronization.

COMPUTATION THREAD

PREFETCH THREAD

```
void main(void){
    int fp, int data, int index;
XX    /* FILE *config_fp; */
## C1. create_prefetch_thread(
        (void *)prefetch_function));
    C2. fp = open("mydata.dat", O_RDONLY);
        /* send the fp to prefetch thread */
## C3. send_fileptr(fp);
XX C4. /* config_fp = fopen("config.dat", "r"); */
XX C5. /* fseek(config_fp, 10, SEEK_SET); */
XX C6. /* fscanf(config_fp, "%d", &index); */
        /* wait for data from prefetch thread */
## C7. receive_fscanf("%d", &index);
    C8. lseek(fp, index*4, SEEK_SET);
    C9. read(fp, &data, 4);
    C10. data = data + 10;
    C11. close(fp);
}
```

```
void prefetch_function(void){
    int fp, int index;
    FILE *config_fp;

        /* receive the file pointer */
$$ P1. receive_fileptr(&fp);
$$ P2. inform_open(fp);
    P3. config_fp = fopen("config.dat", "r");
        /* I/O */
    P4. fseek(config_fp, 10, SEEK_SET);
        /* read and send to computation thread */
    P5. send_fscanf(config_fp, "%d", &index);

    P6. prefetch_lseek(fp, index*4, SEEK_SET);
    P7. prefetch_read(fp, 4);

$$ P8. inform_close(fp)
}
```

Figure 3: Data sharing between the computation and the prefetch thread. Both *fp* and *index* local variables need to be shared between the two threads.

4 Generation of Prefetch Thread

This section describes how AASFP extracts the I/O related code from a given program to form a prefetch thread for use at run time.

4.1 Intra-Procedural Dependency Analysis

The goal of intra-procedure dependency analysis is to identify, inside a procedure, all the variables and statements that disk access statements depend on. Let *related_set(x)* of a variable *x* be the set of statements that are involved, directly or indirectly, in generating the value of *x* in a procedure. We use a simple and conservative approach as follows to compute *related_set(x)*:

1. All the statements that directly update the variable *x*, i.e., those that define *x*, are included in *related_set(x)*.
2. Compute the set *A* that contains all the variables used in the statements in *related_set(x)*. Then for each variable *a* $\in A$, include *related_set(a)* into *related_set(x)*.

Note that in a block structured language like C, special care should be taken to restrict the computation of *related_set(x)* in Step 1 within the scope of the declaration of the variable *x*. For example in the code fragment below, lines 4 and 5 should not be included in *related_set* of the variable *i* on line 7.

```
1. void main(void){
2.     int i;
3.     i = 5;
4.     { int i;
5.         i = 6;
6.     }
7.     lseek(fp,i,SEEK_SET);
8. }
```

The above algorithm for computing *related_set(x)* is conservative and simple to implement. However, it might include some redundant definitions of a variable which never reaches any disk I/O statement. Since the generated source code will go through a final compilation phase, we expect that the compiler could eliminate these redundant statements with a more detailed data flow analysis [1].

Given this algorithm to identify related set, we use the following algorithm to analyze a procedure and

deduce the corresponding prefetch thread:

1. Include the disk access calls in the original program that operate on prefetchable files in *PT*, the set of statements that are I/O related. For each variable *x* that appears in the disk access statements, mark it as I/O related, compute *related_set(x)*, and include the result into *PT*.
2. For each flow-control statement, e.g., *for*, *while*, *do-while*, *if-then-else*, *case*, if there is at least one member of *PT* inside its body, mark all the variables used in the boolean expression of the flow-control statement as I/O related. For each such variable *a*, compute *related_set(a)*, and include it in *PT*. Repeat this step until *PT* stops growing in terms of program size.
3. Include into *PT* the declaration statements of all variables that are marked as I/O related.
4. Insert into *PT* the necessary synchronization calls, add *send_XXX* and *receive_XXX* calls to transfer data between the threads, and rename shared global variables to avoid simultaneous updates from both threads.
5. Finally, if a procedure does not contain any I/O related statements after the algorithm completes, then remove its statements from *PT*.

The above algorithm executes on each procedure iteratively until the resulting I/O-related variable set converges.

4.2 Inter-Procedural Dependency Analysis

To generate the prefetch thread for an application program that contains multiple procedures, inter-procedural dependency analysis is required. It propagates the information on whether a variable is I/O related through procedure call arguments, return values, and global variables. This propagation proceeds as follows:

1. For each procedure *P*, let $Q(x_1, x_2, \dots, x_n)$ be one of the procedures that *P* calls with actual parameters (y_1, y_2, \dots, y_n) . If any actual parameter y_i is I/O related in *P*, and it is a pointer

(i.e., the value it points to can be changed inside *Q*), then mark the object x_i points to in *Q* as I/O related. If *P* stores the return value from *Q* in variable *a*, and *a* is I/O related in *P*, then the variable in *Q* corresponding to the return value is considered I/O related.

2. For each procedure $P(y_1, y_2, \dots, y_n)$, let *R* be a procedure that calls *P* with actual parameters $P(z_1, z_2, \dots, z_n)$. If a formal parameter y_i is I/O related in *P*, its corresponding actual parameter z_i is considered I/O related in *R*.
3. All global variables that are I/O related are I/O related within all procedures.

The above algorithm only needs to be applied to the function call graph once if there are no recursive function calls. If recursion occurs, the algorithm may need to be applied more than once until the extracted code converges.

4.3 Limitations

There are some limitations on the extraction of I/O related code. Currently we do not support multi-threaded applications. The inter-procedural analysis is also not implemented yet as all our test applications have their computations performed in one procedure. Memory Mapped I/O is not dealt with now, but theoretically it might be done. First, we could add more parsing work to identify if there is a *mmap* system call with the protection flag set to *PROT_READ*. Second, we could keep track of the later memory accesses related to the returning address by *mmap*. Third, some extra synchronizations should be done to make sure that the prefetch thread runs ahead, pre-maps the address, and passes the address to the computation thread later.

Currently the input programs are assumed to be written in ANSI C, therefore the other benchmarks that we could test are rather limited.

For simplicity, we sometimes may sacrifice the accuracy in terms of granularity of identifying an I/O related object (variable). For example, if there exists an array of (structured) objects, where in fact only one element of them is really I/O related. For an easier implementation, we would classify this entire array as I/O related.

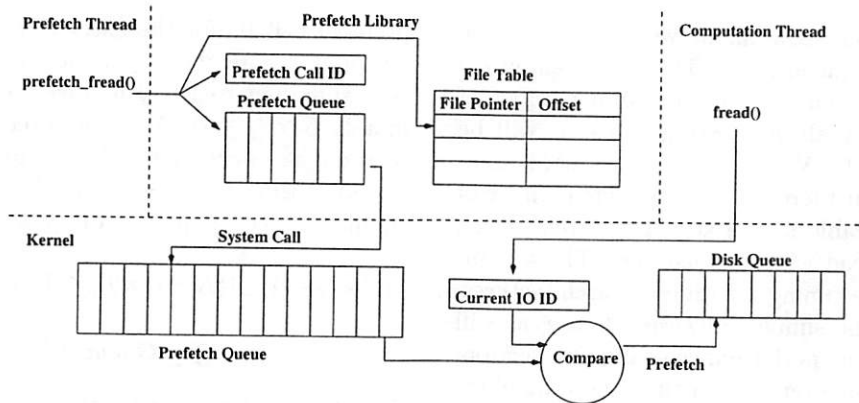


Figure 4: Software Architecture of AASFP's Run-Time System

5 Run-Time System

Figure 4 depicts the software architecture of AASFP's run-time system, which consists of a user-level prefetch library and a kernel component.

5.1 Prefetch Library

The disk I/O calls that the prefetch thread makes are serviced by the prefetch library, which maintains a prefetch queue that stores the addresses of the file blocks to be prefetched. Each prefetch queue entry is a tuple of the form { file ID, block number, prefetch call ID }. The file ID and the block number together uniquely identify the logical disk block to be prefetched, and the prefetch call ID identifies the prefetch call that initiated the corresponding disk access. The initial value of the prefetch call ID is 0. It is incremented every time a prefetch call is made. The library also maintains a file table to keep track of the current offset corresponding to every prefetchable file descriptor so that it can calculate the logical block number for `prefetch_read` and `prefetch_lseek` calls.

Given a prefetch call, the library first assigns the current prefetch call ID to this call. Second, it increments the current prefetch call ID. Third, it inserts an entry into the prefetch queue after calculating its target logical block ID. Finally, it updates the current offset of the accessed file in the file table. When the prefetch queue becomes full, the library makes a system call that we added to copy the prefetch queue entries into the kernel.

When an application makes a `create_prefetch_thread()` call, the library forks off a new thread as the prefetch thread, and informs the kernel about the process ID of both the computation and the prefetch thread using an added system call. This helps the kernel to identify the process ID of the prefetch thread given a computation thread, and vice versa. In addition, the prefetch library registers the file pointers of all prefetchable files with the kernel through another added system call, so that the kernel can take appropriate action regarding prefetching for those file pointers.

5.2 Kernel Support

The modification of the Linux kernel is mainly to ensure that the prefetch thread will be scheduled ahead of but not too far ahead of the computation thread. When the prefetch library of an AASFP application registers with the kernel the process ID of the application's prefetch and computation threads, the kernel allocates a prefetch queue for that application in the kernel address space. When an application's prefetch thread informs the kernel that its user-level prefetch queue is full, the kernel copies them to the application's kernel-level prefetch queue. If there is not enough space in the kernel-level prefetch queue, the prefetch thread is put to sleep. The kernel wakes up the prefetch thread only when there are enough free entries in the kernel's prefetch queue. The size of the kernel prefetch queue is larger than the prefetch library's queue to avoid unnecessary stalls. Note that the prefetch calls in the prefetch thread simply *prepare* disk prefetch requests, but do not actually *initiate* physical prefetch

operations. In our experiment we use 300 for the prefetch library queue size. This is an empirical value which is about 1/10 of the total number of pages accessed by all our testing cases, as will be shown in Table 1. We also set a threshold number 64, and when there are at least this number of free entries available in the kernel prefetch queue, the prefetch thread will be woken up. These numbers may require tuning for different architectures. One could use the simple *Backward 1* case, as will be described in the performance evaluation section, as a good guideline on how to tune these numbers effectively.

For prefetchable files, the AASFP kernel turns off Linux's sequential prefetching mechanism and supports application-specific prefetching. Whenever an AASFP application's computation thread makes a disk access call, the kernel first satisfies this access with data already prefetched and stored in the file buffer, then performs asynchronous disk read for a certain number of requests in the kernel-level prefetch queue. That is, physical disk prefetching is triggered by disk accesses that the computation thread makes. This scheme works well for applications with periodic I/O calls. However, if an application performs a long computation followed by a burst of I/O, physical disk prefetch operations may be invoked too late to mask all disk I/O delay. Therefore AASFP uses a timer-driven approach to schedule disk prefetch operations. That is, every time Linux's timer interrupt occurs (roughly every 10ms), the CPU scheduler will assign a higher priority to the prefetch thread so that the prefetch thread can get scheduled sooner in the near future if it does not find any entry in the kernel-level prefetch queue. Furthermore it will check whether there are prefetch entries in the kernel-level prefetch queue that should be moved to the disk queue according to an algorithm described next. Before a request in the kernel-level prefetch queue is serviced, the kernel checks whether this request is still valid by comparing its prefetch call ID with the number of disk I/O calls that the computation thread has made up to that point. If the prefetch entry's call ID is smaller, the entry is invalid and the kernel just deletes it. For a valid entry, the kernel dynamically determines whether to service that entry at that moment. To make this decision, the kernel maintains the current number of entries in the disk queue (K), the average time taken to service a disk request (T), and the average computation time between two consecutive disk I/O calls for the application (C). Suppose at time t , the current disk I/O call's ID is i , and the

prefetch call ID for the entry is j . Then, the time available before the application accesses the block corresponding to the j th prefetch call is approximately $C \times (j - i)$. A prefetch request that is sent to the disk queue at t will be expected to be completed at time $t + (K + 1) \times T$. Therefore the kernel should service the prefetch request only if

$$(C \times (j - i)) - ((K + 1) \times T) \leq \text{TimeThreshold} \quad (1)$$

$$(j - i) \leq \text{QueueThreshold} \quad (2)$$

The first term ensures that the disk block is prefetched before it is needed, and the second term ensures that there are not too many prefetch blocks in the buffer cache. Keeping the file buffer cache from being over-committed is essential to prevent interference between current and future working sets. Both *QueueThreshold* and *TimeThreshold* are empirical constants that need to be fine-tuned by the users based on hardware configurations and workload characteristics based on system performance; this tuning needs to be done only once per different architecture.

6 Performance Evaluation

This section describes how we conducted our experiments and evaluates the performance results between AASFP and the original Linux on several benchmarks.

6.1 Methodology

We have successfully implemented a fully operational AASFP prototype under Linux 2.0.34. For the source-to-source translator, we did not modify Gcc, but built a parser of our own, which currently only accepts programs written in ANSI C. The parser itself consists of 2.5K lines of code and the I/O extraction part about 3K lines of code. The modification to the Linux kernel involves about only 500 lines of code and the modification to the device driver code is less than 50 lines; therefore this work should be fairly easy to port to new Linux kernels. To evaluate the prototype's performance, we ran one micro-benchmark and two real media applications, and measured their performance on a 200-MHz PentiumPro machine with 64MByte memory.

Test Case	Scenario Description	# of (4KB) Pages Accessed
<i>Vol Vis 1</i>	16MB, orthographic view, 4KB-block	4096
<i>Vol Vis 2</i>	16MB, non-orthographic view, 4KB-block	3714
<i>Vol Vis 3</i>	16MB, orthographic view, 32KB-block	4096
<i>Vol Vis 4</i>	16MB, non-orthonormal view, 32KB-block	3856
<i>FFT 256K</i>	2MB, 256K points, 4KB-block	2944
<i>FFT 512K</i>	4MB, 512K points, 4KB-block	6400
<i>Forward 1</i>	16MB, read forward, 4KB stride	4096
<i>Backward 1</i>	16MB, read backward, 4KB stride	4096
<i>Forward 2</i>	16MB, read forward, 8KB stride	2048
<i>Backward 2</i>	16MB, read backward, 8KB stride	2048

Table 1: Characteristics of test applications.

The first real-application is a volume visualization program based on the direct ray casting algorithm [27]. The volume data set used here is of the size $256 \times 256 \times 256$ and each data point is one byte. This data set is divided into equal-sized blocks, which is the basic unit of disk I/O operation. The block size can be tuned to exploit the best trade-off between disk transfer efficiency and computation-I/O parallelism. In this experiment, we view that data from different viewing directions and use different block sizes. Results for two block sizes are reported here: 4KB ($16 \times 16 \times 16$) and 32KB ($32 \times 32 \times 32$). For non-orthonormal viewing directions, the access patterns of the blocks are quite random. Therefore it provides a good example showing that the default Linux prefetching algorithm can do little help here.

The second application is an out-of-core FFT program [20]. The original program uses four files for reading and writing. We have modified it to merge all the reads and writes into one big file. We have tested the FFT program with 256K points and 512K points of complex numbers; the input file sizes are 2MB and 4MB respectively. Each read/write unit is 4KB bytes.

Table 1 shows the characteristics of different applications we used in this performance evaluation study. *Vol Vis 1*, *Vol Vis 2*, *Vol Vis 3*, and *Vol Vis 4* are four variations of the volume visualization application viewed from different angles with different block sizes. *FFT 256K* and *FFT 512K* are the out-of-core FFT program with different input sizes. *Forward 1*, *Backward 1*, *Forward 2*, and *Backward 2* are variations of a micro-benchmark that emulates the disk access behavior of a digital video player that

supports fast forward and backward in addition to normal playbacks.

6.2 Performance Results and Analysis

Table 2 shows the measured performance of the test cases in Table 1 under generic Linux and when using AASFP. All numbers are the average results of 5 runs. To get correct results, the file system buffer cache must be flushed each time. Instead of rebooting our testing machine each time, we generate a file with random contents and whose size is 128MBytes. Notice this size is bigger than or equal to the size of the system's physical memory size plus the system's swap space size. Therefore, a sequential read through out this file should wipe out all the related content of the previous run. To verify this, we compare the results of *Backward 1* under normal Linux with the machine rebooted each time and with reading the above big file. The numbers are shown in Table 3.

Under Linux, only sequential disk prefetching is supported. Under AASFP, only application-specific disk prefetching is supported. The number within the parenthesis shows the AASFP overhead, which is due to the extra time to run the prefetch thread. This overhead is in general insignificant because the cost of performing computation in the prefetch thread, and the associated process scheduling and context switching is relatively small when compared to the computation overhead. The percentages of disk I/O time that AASFP can mask are listed in the fourth column. This is calculated by taking the ratio between the disk I/O time that is masked and

Test Case	Linux	AASFP (Overhead)	% of Disk I/O Masked	Perf. Improv.	CPU Time	% of Syn. Overhead	% of Com. Overhead
<i>Vol Vis 1</i>	68.95	31.76 (3.59)	62.14%	53.94%	24.47	0.06%	0.18%
<i>Vol Vis 2</i>	83.05	64.95 (3.22)	12.99%	21.56%	15.18	0.76%	0.09%
<i>Vol Vis 3</i>	36.87	31.23 (3.02)	66.61%	15.30%	25.93	0.00%	0.19%
<i>Vol Vis 4</i>	30.99	29.78 (3.02)	30.00%	3.90%	15.46	2.04%	0.20%
<i>FFT 256K</i>	33.42	33.74 (0.00)	0.00%	-0.94%	24.70	2.16%	0.12%
<i>FFT 512K</i>	66.68	67.84 (0.00)	0.00%	-1.71%	54.75	1.35%	0.06%
<i>Forward 1</i>	4.78	4.76 (0.00)	0.54%	0.42%	0.48	0.21%	0.21%
<i>Backward 1</i>	52.54	7.63 (0.00)	84.75%	85.48%	0.48	0.65%	0.13%
<i>Forward 2</i>	4.61	4.84 (0.00)	0.00%	-4.75%	0.48	0.00%	0.21%
<i>Backward 2</i>	25.02	6.19 (0.00)	78.40%	75.26%	0.48	0.32%	0.16%

Table 2: The overall performance measurements of the test applications under Linux and under AASFP. All reported measurements are in seconds or percentage. AASFP overhead (included in the AASFP time, and is also shown within the parenthesis) is mainly the time to execute the prefetch thread, which does not exist in the conventional approaches. Percentages of Masked I/O shows the percentages of disk I/O time that AASFP can effectively mask.

the total disk I/O time without prefetching.

The fifth column in Table 2 shows performance improvement column of AASFP over Linux. For reference, we also list the CPU Time, which is the pure computation time of the application (i.e., excluded I/O), in the sixth column. The overhead due to synchronization, as explained in Section 3.2 is shown in the seventh column here. The last column shows the associated compilation overhead of AASFP to extract the prefetch thread from each program.

For the volume visualization application with a 4-KByte block size, AASFP achieves 54% and 22% overall performance improvement for orthonormal and non-orthonormal viewing directions, respectively. There is not much performance gain for the cases that use 32-KByte block size. Retrieving 32-KByte blocks corresponds to fetching eight 4K pages consecutively. There is substantial spatial locality in this access pattern, which detracts the relative performance gain from AASFP. This also explains why the generic Linux prefetching is comparable to AASFP when 32-KByte blocks are used.

For the out-of-core FFT apparently there is no significant performance improvement. This is due to its extremely sequential accessing patterns. Although FFT is well known by its butterfly algorithmic structure, which suggests random disk access patterns, a more careful examination revealed that not only out-of-core FFT, but also many other out-of-core applications exhibit sequential disk access

patterns. Nevertheless our results show that even under such fairly regular access patterns AASFP can still provide as good performance as sequential prefetching. This means that AASFP does not mistakenly prefetch something that is unnecessary and eventually hurt the overall performance, and that the prefetch thread does not add any noticeable overhead in this case.

For the micro-benchmark, AASFP provides 86% performance improvement for the *Backward 1* case, which represents the worst case for the sequential prefetching scheme used in Linux. Note that AASFP almost does not lose any performance for the *Forward 1* and *Forward 2* case when compared to Linux. This is the best case for the Linux kernel. The last measurement again demonstrates that AASFP performs as well as generic Linux for sequential access patterns.

Another potential factor that can hurt the prefetching efficiency is synchronization. Too many forced synchronizations sometimes will slow down how far ahead the prefetch thread can go, thus limiting the performance improvement of AASFP. In all the above test cases, the synchronization-related overhead is either non-existent or negligible, because there are neither conditional branches nor user-inputs (for deciding which data file to use), as can be shown in Table 2.

Finally, the associated compilation overhead is also minimal, and furthermore, the prefetch thread code

Reboot	52.31	52.60	52.87	52.47	52.88
Flush	52.40	52.53	52.60	52.53	52.58

Table 3: *The comparison between rebooting a system and reading a gigantic file to flush the file system's buffer cache. Reported numbers are the average times of 5 runs of the Backward 1 case under Linux.*

extraction needs to be done only once in a preprocessing mode. This suggests not only the simplicity of AASFP but also its applicability to other similar programs.

7 Conclusion and Future Work

We have designed, implemented and evaluated an automatic application-controlled file prefetching system called AASFP that is particularly useful for multimedia applications with irregular disk access patterns. It is automatic in the sense that the system exploits application-specific disk access patterns for file prefetching without any manual programming. The idea of extracting I/O related code from the original code is very general and we believe it is applicable to other languages such as C++ or Java as well; the required effort to support other languages should also be comparable to this work. The Linux-based AASFP prototype implementation is fully operational and provides up to 54% overall application improvement for a real-world volume visualization application. Currently we are continuing the development of AASFP. We are extending the current prototype to allow multiple I/O-intensive applications to run on an AASFP-based system simultaneously. The key design issue here is to allocate disk resource among multiple processes, depending on their urgency on disk access requirements. We are also building on the AASFP technology to develop a high-performance I/O subsystem for large-scale parallel computing clusters.

Finally we are extending the AASFP prototype to the context of Network File System (NFS), and generalize the application-specific prefetching to a more general concept called *active file server* architecture. Here, in addition to standard file access service, we allow an application to deposit an arbitrary program either to manipulate accessed file data on the fly such as compression or encryption (data plane), or

to exercise different control policies such as prefetching, replacement, and garbage collection (control plane). The active file server architecture significantly enhances the customizability and flexibility of file access, and thus improves both the performance of individual applications and the overall efficiency of the file system. A major research challenge for active file server is the design of a procedural interface for *application program segments* that is both general and efficient enough to accommodate various control plane or data plane processing requirements, and sufficiently rigid to ensure data security and safety. We plan to use AASFP with NFS as a case study to gain some concrete experiences with the architectural design issues associated with active file server. The code of this project will be available for download at the URL: <http://www.ecsl.cs.sunysb.edu/archive.html>.

Acknowledgments

We would like to thank professor Erez Zadok for his numerous suggestions and those anonymous USENIX reviewers for their invaluable comments. This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principle, Techniques, and Tools*. Addison-Wesley Publishing Company, 1988.
- [2] P. Cao et al. Application Controlled File Caching Policies. In *USENIX summer 1994 Technical Conference*, June 1994.
- [3] P. Cao et al. Implementation and Performance of Application-Controlled File Caching. In *First USENIX Symposium on Operating Systems Design and Implementation*, November 1994.

- [4] P. Cao et al. A Study of Integrated Prefetching and Caching Strategies. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.
- [5] F. Chang and G. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.
- [6] K. Curewitz et al. Practical Prefetching via Data Compression. In *ACM Conference on Management of Data*, May 1993.
- [7] D. Duchamp et al. Prefetching Hyperlinks. In *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.
- [8] T. Kimbrel et al. A Trace Driven Comparison for Parallel Prefetching and Caching. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.
- [9] D. Kotz et al. Practical Prefetching Techniques for Parallel File Systems. In *First International Conference on Parallel and Distributed Information Systems*, December 1991.
- [10] H. Lei et al. An Analytical Approach to File Prefetching. In *Proceedings of the 1997 Usenix Annual Technical Conference*, January 1997.
- [11] X. Leroy. The LinuxThreads Library. <http://pauillac.inria.fr/~xleroy/linuxthreads/>.
- [12] M. K. McKusick et al. A Fast File System for UNIX. *ACM Transaction on Computer Systems*, 2(3), August 1984.
- [13] T. C. Mowry et al. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.
- [14] V. Pai et al. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Usenix Annual Technical Conference*, 1999.
- [15] R. H. Patterson. Using Transparent Informed Prefetching to Reduce File System Latency. In *Goddard Conference on Mass Storage Systems and Technologies*, September 1992.
- [16] R. H. Patterson. *Informed Prefetching and Caching*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1997.
- [17] R. H. Patterson et al. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2), April 1993.
- [18] R. H. Patterson et al. Exposing I/O Concurrency with Informed Prefetching. In *3rd International Conference on Parallel and Distributed Information Systems*, September 1994.
- [19] R. H. Patterson et al. Informed Prefetching and Caching. In *15th ACM Symposium on Operating System Principle*, December 1995.
- [20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1997.
- [21] D. Rochberg et al. Prefetching Over a Network: Early Experience with CTIP. *SIGMETRIC Performance Evaluation Review*, 25(3).
- [22] J. Smith et al. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8), August 1986.
- [23] C. D. Tait et al. Intelligent File Hoarding for Mobile Computers. In *Proceedings of the First International Conference on Mobile Computing and Networking*, November 1995.
- [24] A. Tomkins. *Practical and Theoretical Issues in Prefetching*. PhD thesis, Computer Science Department, Carnegie Mellon University, October 1997.
- [25] A. Tomkins et al. Informed Multi-Process Prefetching and Caching. In *ACM International Conference on Measurement and Modeling of Computer Systems*, June 1997.
- [26] J. S. Vitter et al. Optimal Prefetching via Data Compression. In *32nd Annual IEEE Symposium on Foundations of Computer Science*, October 1991.
- [27] C. Yang and T. Chiueh. I/O-Conscious Volume Rendering. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization*, May 2001.

Design and Performance of the OpenBSD Stateful Packet Filter (pf)

Daniel Hartmeier*
Systor AG
dhartmei@openbsd.org

Abstract

With more and more hosts being connected to the Internet, the importance of securing connected networks has increased, too. One mechanism to provide enhanced security for a network is to filter out potentially malicious network packets. Firewalls are designed to provide “policy-based” network filtering.

A firewall may consist of several components. Its key component is usually a packet filter. The packet filter may be stateful to reach more informed decisions. The state allows the packet filter to keep track of established connections so that arriving packets could be associated with them. On the other hand, a stateless packet filter bases its decisions solely on individual packets. With release 3.0, OpenBSD includes a new Stateful Packet Filter (*pf*) in the base install. *pf* implements traditional packet filtering with some additional novel algorithms. This paper describes the design and implementation of *pf* and compares its scalability and performance with existing packet filter implementations.

1 Introduction

The main emphasis of the OpenBSD project is proactive and effective computer security. The integration of a Stateful Packet Filter is an important aspect. Since 1996, Darren Reed’s *IPFilter* has been included in the OpenBSD base install. It was removed after its license turned out to be incompatible with OpenBSD’s goal of providing software that is free to use, modify and redistribute in any way for everyone.

*Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0537.

While an acceptable Open Source license was a prerequisite for any replacement, we used this opportunity to develop a new packet filter that employed optimized data structures and algorithms to achieve good performance for stateful filtering and address translation. The resulting code base is small and thus facilitates future extensions.

The remainder of this paper is organized as follows. Section 2 outlines the design of the packet filter. In Section 3 we compare *pf*’s performance with other packet filters and discuss the results. Section 4 presents future work. Finally, we conclude in Section 5.

2 Design

The Stateful Packet Filter resides in the kernel and inspects every IP packet that enters or leaves the stack. It may reach one of several decisions:

- to pass a packet unmodified or modified,
- to silently block a packet, or
- to reject packet with a response, e.g., sending a TCP reset.

The filter itself consists of two basic elements, the filter rules and the state table.

2.1 Filter rules

Every packet is compared against the filter rule set. The rule set consists of a linked list of rules. Each rule contains a set of parameters that determines the set of packets the rule applies to. The parameters may be the source or destination address, the protocol, port numbers, etc. For a packet that matches

the rule, the specified *pass* or *block* action is taken. *Block* means that the packet is dropped by the filter, and *pass* means that the packet is forwarded to its destination.

During rule set evaluation, the packet is compared against all rules from the beginning to the end. A packet can match more than one rule, in which case the last matching rule is used. This mechanism allows overriding general rules with more specific rules, like blocking all packets first and then passing specific packets. The last matching rule determines if the packet is passed or blocked according to its action field.

A matching rule that has been flagged as *final* terminates rule evaluation for the current packet. The action from that rule is applied to the packet. This prevents *final* rules from being overridden by subsequent rules.

The rule set is organized in a multiple linked list. This allows *pf* to perform automatic optimization of the rule set as discussed in Section 2.8.

2.2 State table

Stateful packet filtering implies that a firewall inspects not only single packets, but also that it knows about established connections. Any rule that passes a packet may create an entry in the state table. Before the filter rule set is evaluated for a packet, the state table is searched for a matching entry. If a packet is part of a tracked connection, it is passed unconditionally, without rule set evaluation.

For TCP, state matching involves checking sequence numbers against expected windows [8], which improves security against sequence number attacks.

UDP is stateless by nature: packets are considered to be part of the same connection if the host addresses and ports match a state entry. UDP state entries have an adaptive expiration scheme. The first UDP packet could either be a one-shot packet or the beginning of a UDP pseudo-connection. The first packet will create a state entry with a low timeout. If the other endpoint responds, *pf* will consider it a pseudo-connection with bidirectional communication and allow more flexibility in the duration of the state entry.

ICMP packets fall into two categories: ICMP error messages which refer to other packets, and ICMP queries and replies which are handled separately. If an ICMP error message corresponds to a connection in the state table, it is passed. ICMP queries and replies create their own state, similar to UDP states. As an example, an ICMP echo reply matches the state an ICMP echo request created. This is necessary so that applications like *ping* or *traceroute* work across a Stateful Packet Filter.

pf stores state entries in an AVL tree. An AVL tree is a balanced binary search tree. This container provides efficient search functionality which scales well for large trees. It guarantees the same $O(\log n)$ behavior even in worst case. Although alternative containers like hash tables allow searches in constant time, they also have their drawbacks. Hash tables have a fixed size by nature. As the number of entries grows, collisions occur (if two entries have the same hash value) and several entries end up in the same hash bucket. An attacker can trigger the worst case behavior by opening connections that lead to hash collisions and state entries in the same hash bucket. To accommodate this case, the hash buckets themselves would have to be binary search trees, otherwise the worst case behavior allows for denial of service attacks. The hash table would therefore only cover a shallow part of the entire search tree, and reduce only the first few levels of binary search, at considerable memory cost.

2.3 Network address translation

Network address translation (NAT) is commonly used to allow hosts with IP addresses from a private network to share an Internet connection using a single route-able address. A NAT gateway replaces the IP addresses and ports from packet that traverse the gateway with its own address information. Performing NAT in a Stateful Packet Filter is a natural extension, and *pf* combines NAT mappings and state table entries. This is a key design decision which has proved itself valuable.

The state table contains entries with three address/port pairs: the internal, the gateway and the external pair. Two trees contain the keys, one sorted on internal and external pair, the other sorted on external and gateway pair. This allows to find not only a matching state for outgoing and incoming packets, but also provides the NAT mapping in the

same step without additional lookup costs. *pf* also supports port redirection and bidirectional translation. Application-level proxies reside in userland, e.g., ftp-proxy which allows active mode FTP for clients behind a NAT gateway.

2.4 Normalization

IP normalization removes interpretation ambiguities from IP traffic [5]. For example, operating systems resolve overlapping IP fragments in different ways. Some keep the old data while others replace the old data with data from a newly arrived fragment. For systems that resolve overlaps in favor of new fragments, it is possible to rewrite the protocol headers after they have been inspected by the firewall.

While OpenBSD itself is not vulnerable to fragmentation attacks [3, 4], it protects machines with less secure stacks behind it. Fragments are cached and reassembled by *pf* so that any conflict between overlapping fragments is resolved before a packet is received by its destination [2].

2.5 Sequence number modulation

pf can modulate TCP sequence numbers by adding a random number to all sequence numbers of a connection. This protects hosts with weak sequence number generators from attacks.

2.6 Logging

pf logs packets via bpf [6] from a virtual network interface called *pflog0*. This allows all of the existing network monitoring applications to monitor the *pf* logs with minimal modifications. *tcpdump* can even be used to monitor the logging device in real time and apply arbitrary filters to display only the applicable packets.

2.7 States vs. rule evaluation

Rule set evaluation scales with $O(n)$, where n is the number of rules in the set. However, state lookup

scales with $O(\log m)$, where m is the number of states. The constant cost of one state key comparison is not significantly higher than the comparison of the parameters of one rule. This means that even with smaller rule sets, filtering statefully is actually more efficient as packets that match an entry in the state table are not evaluated by the rule set.

2.8 Transparent rule set evaluation optimization

pf automatically optimizes the evaluation of the rule set. If a group of consecutive rules all contain the same parameter, e.g., "source address equals 10.1.2.3," and a packet does not match this parameter when the first rule of the group is evaluated, the whole group of rules is skipped, as the packet can not possibly match any of the rules in the group.

When a rule set is loaded, the kernel traverses the set to calculate these so-called *skip-steps*. In each rule, for each parameter, there is a pointer set to the next rule that specifies a different value for the parameter.

During rule set evaluation, if a packet does not match a rule parameter, the pointer is used to skip to the next rule which could match, instead of the very next rule in the set.

The *skip-steps* optimization is completely transparent to the user because it happens automatically without changing the meaning of any rule set.

The performance gain depends on the specific rule set. In the worst case, all *skip-steps* have a length of one so that no rules can be skipped during evaluation, because the parameters of consecutive rules are always different. However, even in the worst case, performance does not decrease compared to an unoptimized version.

An average rule set, however, results in larger *skip-steps* which are responsible for a significant performance gain. The cost of evaluating a rule set is often reduced by an order of magnitude, as only every 10th to 100th rule is actually evaluated.

Firewall administrators can increase the likelihood of *skip-steps* optimizations and thereby improving the performance of rule evaluation by sorting blocks of rules on parameters in a documented order.

Automatically generated groups of rules are already sorted in optimal order, e.g., this happens when one rule contains parameter lists that are expanded internally to several new rules.

3 Performance evaluation

We evaluate the performance of the packet filter by using two hosts with two network interface cards each, connected with two crossover Cat5 cables, in 10baseT unidirectional mode.

The *tester* host uses a libnet program to generate TCP packets as ethernet frames. They are sent through the first interface to the *firewall* host and captured as ethernet frames on the second interface of the *tester* using libpcap. The two hosts do not have any other network connections.

The firewall is configured to forward IP packets between its interfaces, so that the packets sent by the tester are forwarded through the other interface back to the tester.

The firewall is an i386 machine with a Pentium 166 MHz CPU and 64 MB RAM; the tester is a faster i386 machine. All four network interface cards are identical NetGear PCI cards, using the sis driver.

Arp table entries are static, and the only packets traversing the wires are the packets generated by the tester and forwarded back by the firewall.

The generated packets contain time stamps and serial numbers, that allow the tester to determine latency and packet loss rate.

The tester is sending packets of defined size (bytes/packet, including ethernet header and checksum) at defined rates (packets/s), and measures the rate of received packets, the average latency and loss rate (percentage of packets lost). For each combination, the measuring period is at least ten seconds. Latency is the average for all packets returned to the tester during the measuring period. Lost packets are not counted towards latency.

Successively, the following tests are repeated with the same firewall running OpenBSD 3.0 with *pf*, OpenBSD 3.0 with *IPFilter* and GNU/Linux Red-

Hat 7.2 with *iptables*.

3.1 Unfiltered

The first test is run without interposing the packet filter into the network stack on the firewall.

Figure 1 shows that both the tester and the firewall are able to handle packets at the maximum frame rate [1] for all packet sizes of 128 bytes and above. All further tests are done using packet sizes of either 128 or 256 bytes. The degradation for the GNU/Linux machine seems slightly worse than for the OpenBSD machine.

3.2 Stateless filtering with increasing rule set size

In the second test, the packet filter is enabled and the size of the filter rule set is increased repeatedly. The rules are chosen so that the packet filter is forced to evaluate the entire rule set and then pass each packet without creating state table entries. The generated packets contain random port numbers to defeat any mechanisms used by the packet filter to cache rule set evaluations.

Figures 2, 3 and 4 show throughput, latency and loss depending on sending rate, for a set of 100 rules, using 256 byte packets. *Iptables* outperforms both *pf* and *IPFilter* in this test. It has a higher maximum throughput and lower latency compared to the other two packet filters.

Each packet filter has a distinct maximum lossless throughput. If the sending rate exceeds this maximum, latency increases quickly and packet loss occurs. For all three filters, latency is nearly identical below the maximum lossless rate. When the sending rate is increased beyond the point where loss occurs, throughput actually decreases. In such overloaded condition, all three packet filter consume all CPU resources and the console becomes unresponsive. After the sending rate is lowered below the maximum throughput rate, each one of them recovers quickly.

The test is repeated with various rule set sizes between one and ten thousand. For each rule set size, the maximum throughput rate possible without packet loss is noted. The results are shown in

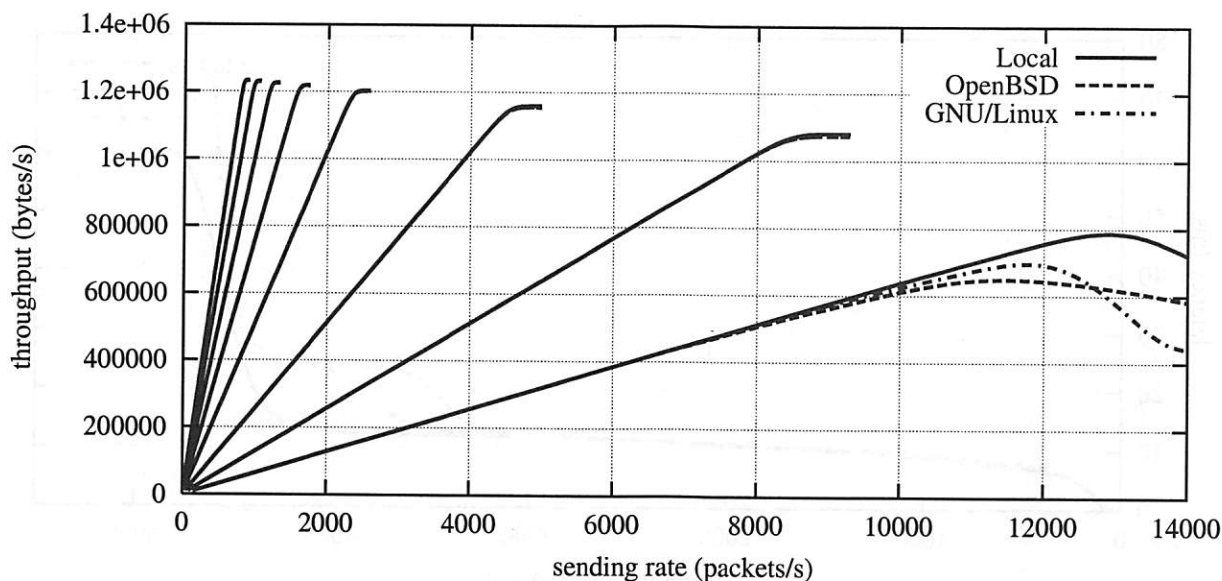


Figure 1: Unfiltered

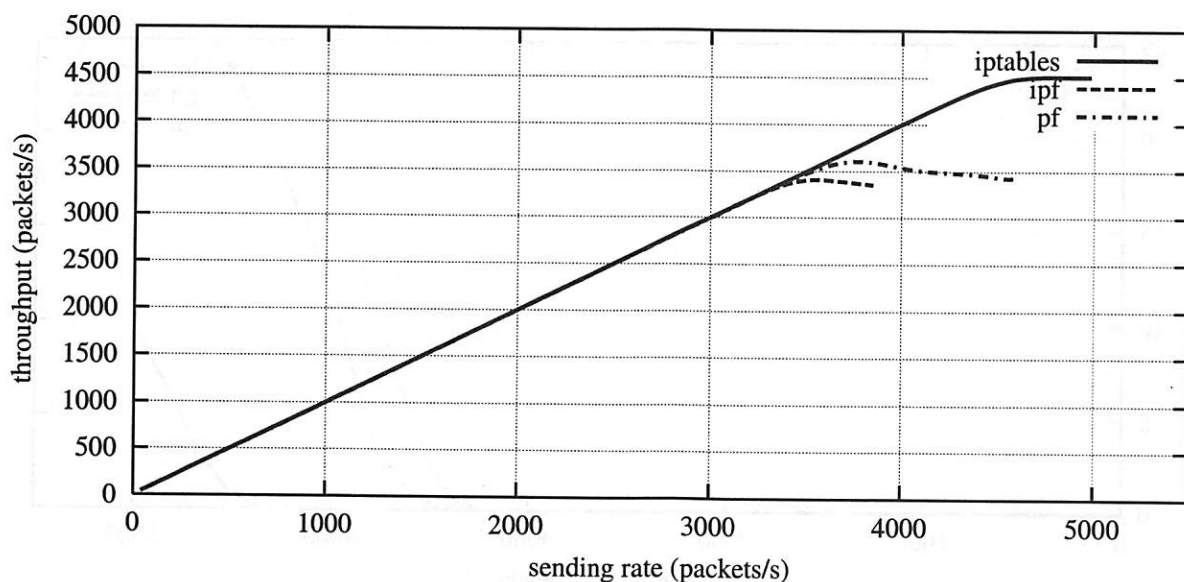


Figure 2: Stateless filtering with 100 rules (throughput)

Figure 5 as a function of the rule set size. Both *pf* and *IPFilter* evaluate the rule set twice for each packet, once incoming on the first interface and once outgoing on the second interface. *Iptables*' performance advantage is due to the fact that it evaluates the rule set only once by using a forwarding chain. The forwarding chain evaluates the rules set based on a packet's complete path through the machine.

3.3 Stateful filtering with increasing state table size

The third test uses a rule set that contains only a single rule to pass all packets statefully, *i.e.*, new state is created for packets that do not belong to any connection tracked by the existing state. To prime the state table, the tester establishes a de-

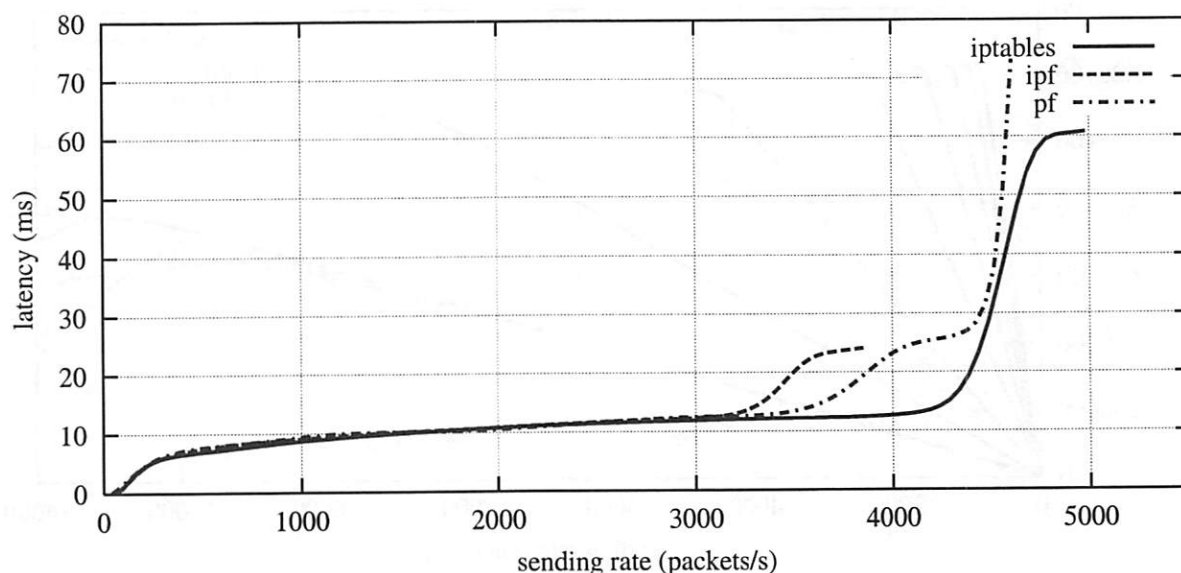


Figure 3: Stateless filtering with 100 rules (latency)

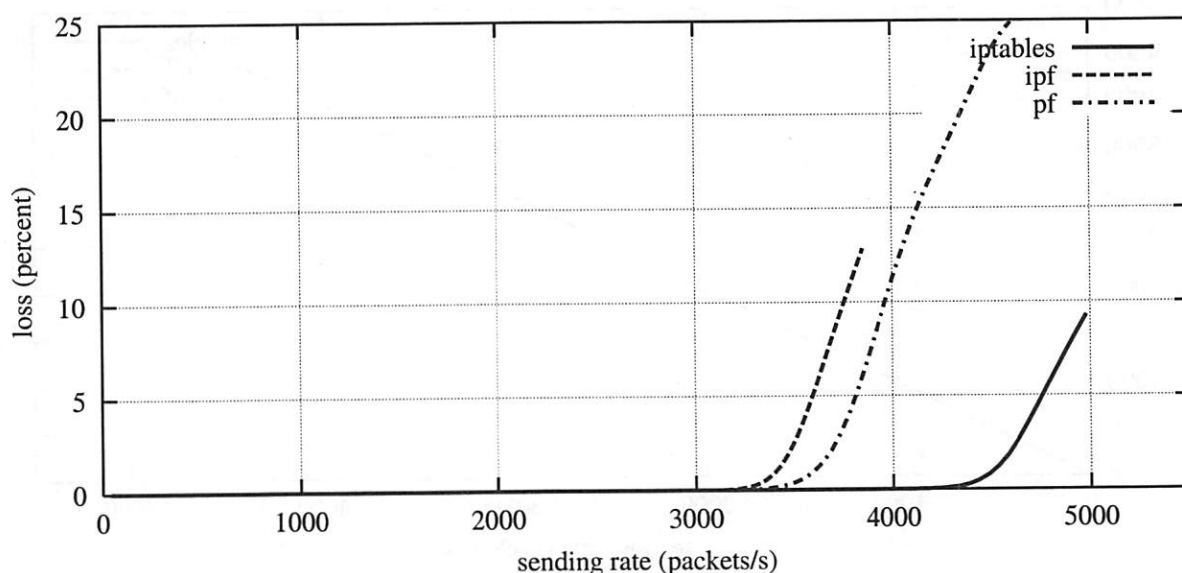


Figure 4: Stateless filtering with 100 rules (loss)

finer number of connections with itself by completing TCP handshakes. After all connections are established, the tester sends random packets matching the established connections with uniform distribution. During the entire test, all state table entries are used, no entries time out and no new entries are added. *Iptables* is not included in the stateful tests as it does not perform proper state tracking as explained below.

Figure 6 compares the throughput in relation to the sending rate for stateful filtering with twenty thousand state entries. Both *pf* and *IPFilter* exhibit the same behavior when overloaded. However, *IPFilter* reaches overload at a packet rate of about four thousand packets per second whereas, *pf* does not reach overload until about six thousand packets per second.

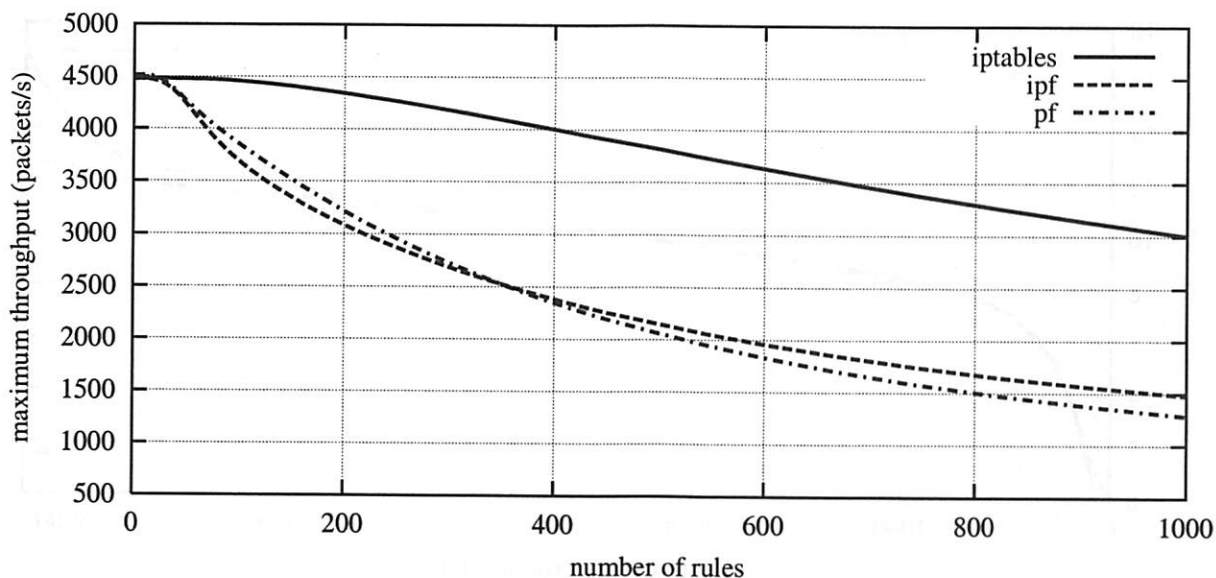


Figure 5: Maximum throughput with increasing number of rules

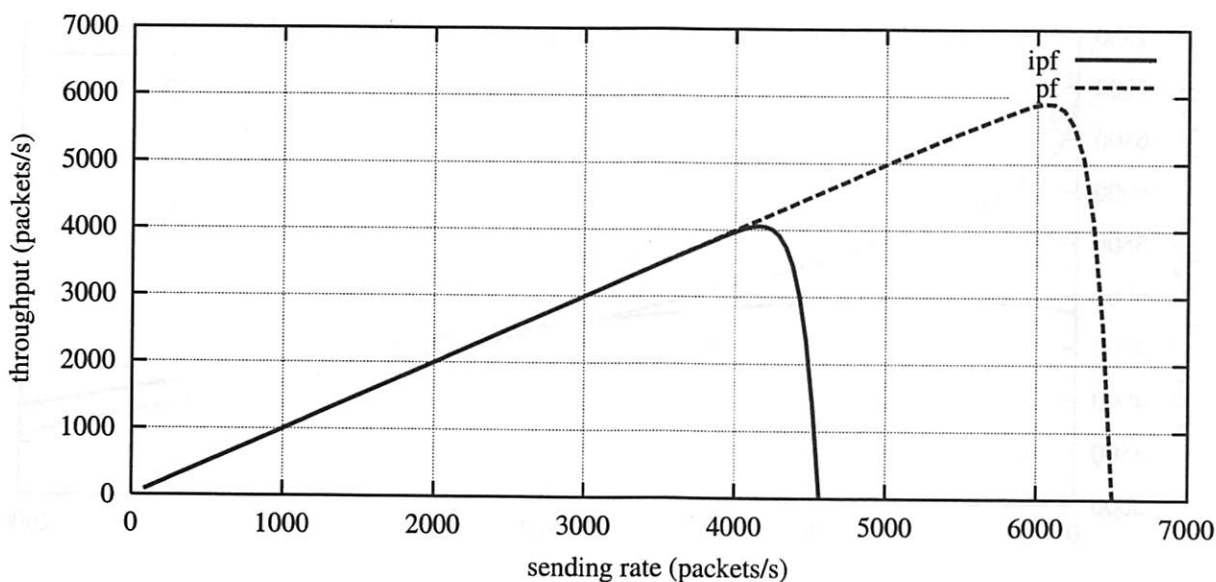


Figure 6: Stateful filtering with 20000 state entries (throughput)

The latency comparison for this test is shown in Figure 7. The latency increases as expected when the packet filters reach overload.

Similarly to the second test, the procedure is repeated for various state table sizes between one and twenty five thousand. Figure 8 shows the maximum lossless throughput rate as a function of the state table size. We notice that *pf* performs significantly

better than *IPFilter* when the size of the state table is small. At ten thousand state table entries, *IPFilter* starts to perform better than *pf*, but the difference is not significant.

We expected Figure 8 to show the $O(1)$ behavior of hash table lookups for *IPFilter* and $O(\log n)$ of tree lookups for *pf*. However, it seems that the constant cost of hash key calculation is relatively high,

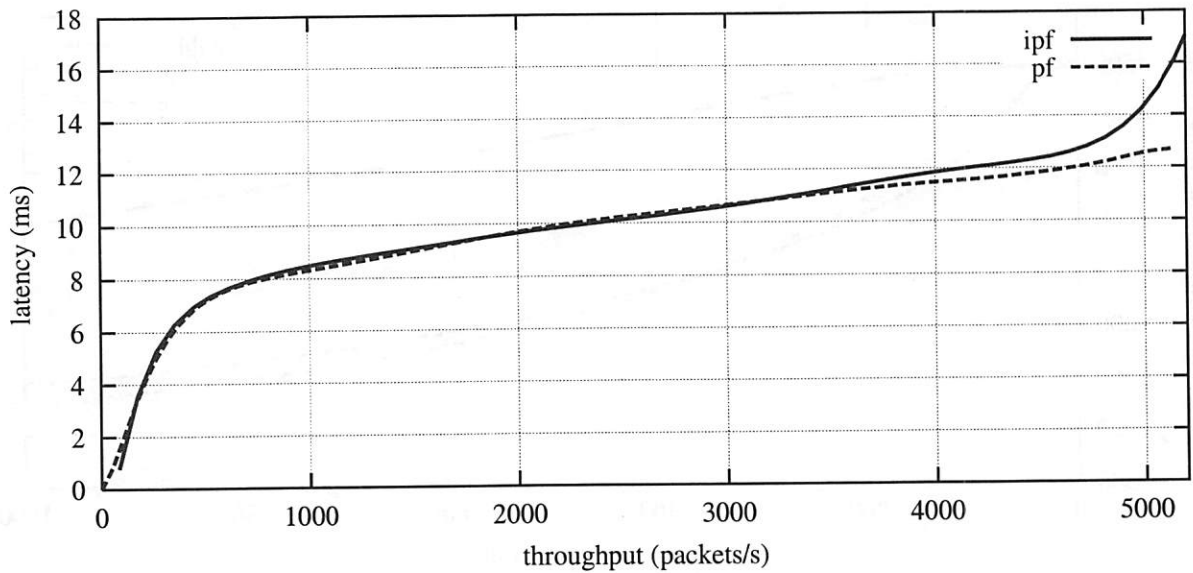


Figure 7: Stateful filtering with 20000 state entries (latency)

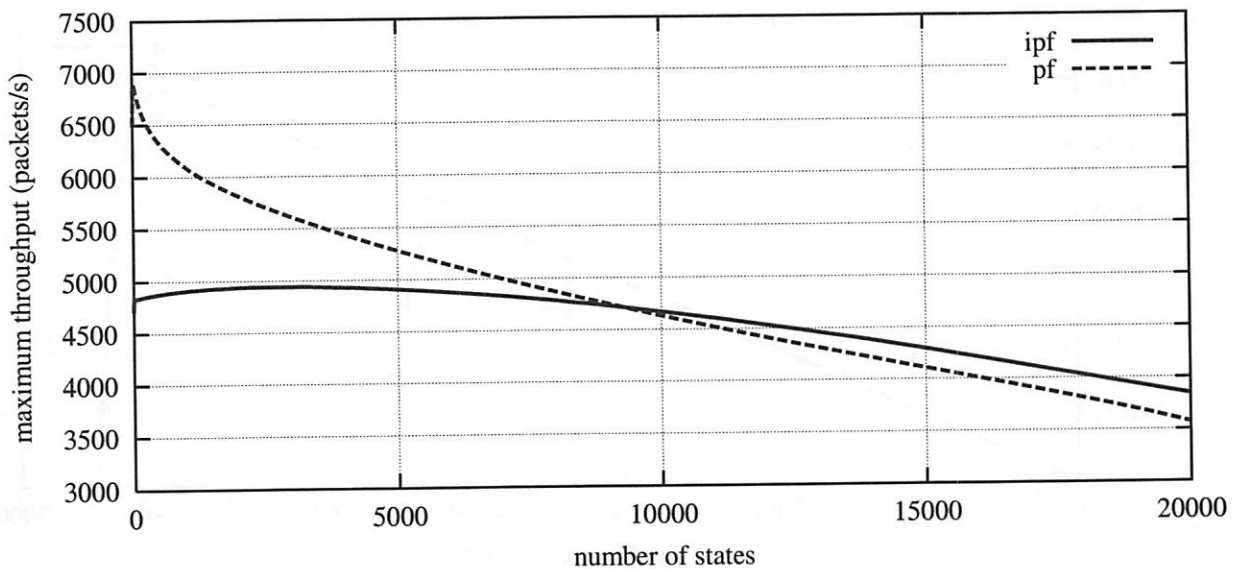


Figure 8: Maximum throughput with increasing number of states

and *IPFilter* still depends on n for some unknown reason.

Iptables has not been included in this benchmark because it does not do stateful filtering comparable to *pf* and *IPFilter*. The version of *iptables* that we tested employs *connection tracking* without any sequence number analysis for packets outside of the initial TCP handshake. While this is unsurprisingly

faster, it would be an unfair performance comparison. There is a patch for *iptables* that adds sequence number checking, but it is still beta and is not included in the GNU/Linux distribution used for testing.

3.4 Discussion

The stateless benchmark indicates that rule set evaluation is very expensive in comparison to state table lookups. During the initial tests, *pf* was considerably slower in evaluating rules than *IPFilter*.

The slower performance is explained by the fact that *pf* used to evaluate the rule set three times for every packet that passes an interface: twice to look for *scrub* rules which determine whether IP and TCP normalization should be performed and once to look for *pass* and *block* rules.

This problem can be rectified by a simple optimization. It is sufficient to add two additional *skip-steps* for rule types *scrub* versus *pass/block* and the direction *in* versus *out*. This change which is now part of OpenBSD 3.1 improves the performance of *pf*'s rule set evaluation considerably, as shown in Figure 9.

The benchmarks measure only how packet filter performance scales for extreme cases to show the behavior of rule set evaluation and state table lookup algorithms, e.g., completely stateless and when all packets match state. In real-life, a firewall will perform different mixtures of these operations, as well as other operations that have not been tested, like creation and removal of state table entries and logging of blocked packets.

Also, real-life packets rarely require a complete evaluation of the rule set. All tested packet filters have mechanisms to reduce the number of rules that need to be evaluated on average. *Iptables* allows the definition of and jumps to *chains* of rules. As a result, the rule set becomes a tree instead of a linked list. *IPFilter* permits the definition of rule *groups*, which are only evaluated when a packet matches a *head* rule. *pf* uses *skip-steps* to automatically skip rules that cannot apply to a specific packet. In summary, *iptables* perform the best for stateless rules and *pf* performs the best when using stateful filtering.

4 Future work

There are still several areas in which *pf* may be improved. Plans for future work include among other things the following:

- application level proxies for additional protocols,
- authentication by modifying filter rules to allow network access based on user authentication,
- load-balancing, e.g., redirections translating destination addresses to a pool of hosts to distribute load among multiple servers,
- fail-over redundancy in which one firewall replicates state information to a stand-by firewall that can take over in case the primary firewall fails and
- further TCP normalization, e.g., resolving overlapping TCP segments, as described in the traffic normalization paper by Handley *et al.* [5].

5 Conclusions

This paper presented the design and implementation of a new Stateful Packet Filter and compared its performance with existing filters. The key contributions are an efficient and scalable implementation of the filter rule set and state table, automatic rule set optimization and unique features such as normalization and sequence number modulation.

The benchmarks show that the lower cost of state table lookups compared to the high cost of rule set evaluations justify creating state for performance reasons. Stateful filtering not only improves the quality of the filter decisions, it effectively improves filtering performance.

The new Stateful Packet Filter is included in OpenBSD 3.0 released in November 2001. The source code is BSD licensed and available in the OpenBSD source tree repository [7].

6 Acknowledgements

The OpenBSD packet filter is a collaborative work, and many developers have contributed code, ideas and support. In particular, I'd like to thank Artur Grabowski, Bob Beck, Dug Song, Jason Ish, Jason Wright, Jun-ichiro itojun Hagino, Kenneth R. Westerback, Kjell Wooding, Markus Friedl, Mike

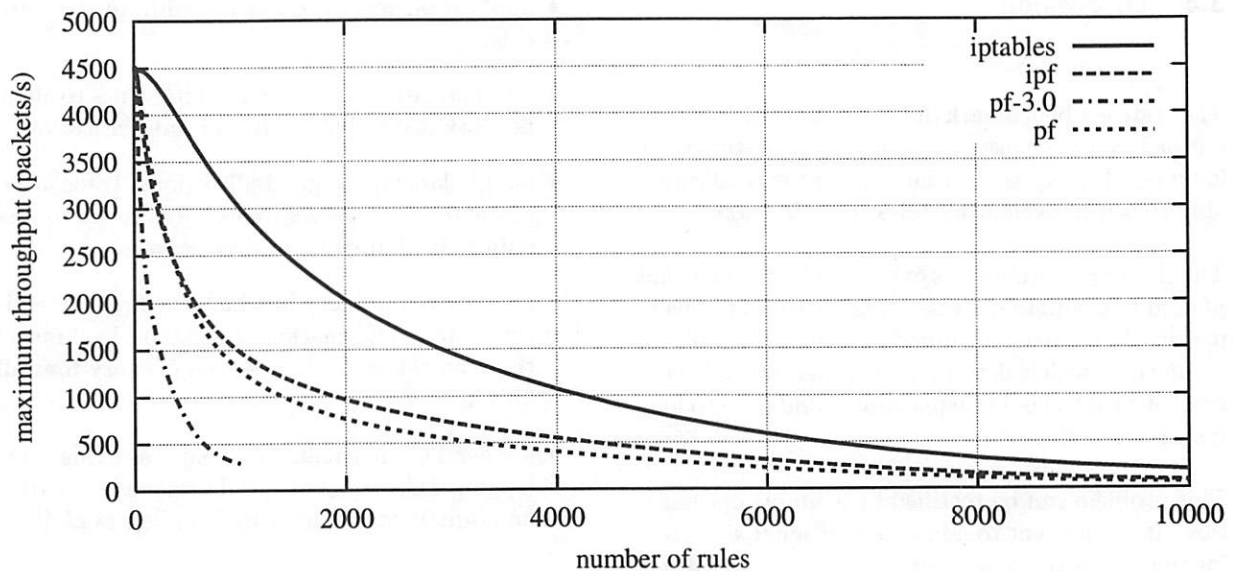


Figure 9: Maximum throughput with increasing number of rules

Frantzen, Niels Provos, Ryan McBride, Theo de Raadt, Todd C. Miller and Wim Vandeputte. Most of the equipment and time used to write this paper was funded by Systor AG.

References

- [1] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. Internet RFC 2544, March 1999.
- [2] David Watson Farnam Jahanian, G. Robert Malan and Paul Howell. Transport and application protocol scrubbing. In *Proc. Infocomm 2000*, 2000.
- [3] Patricia Gilfeather and Todd Underwood. Fragmentation made friendly. In <http://www.cs.unm.edu/~maccabe/SSL/frag/FragPaper1/Fragmentation.html>.
- [4] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. In *WRL Technical Report 87/3*, December 1987.
- [5] C. Kreibich M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium 2001*, 2001.
- [6] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX Conference, San Diego, CA*, January 1993.
- [7] The OpenBSD project. <http://www.openbsd.org/>.
- [8] Guido van Rooij. Real stateful tcp packet filtering in ip filter. In <http://www.madison-gurkha.com/publications/tcp-filtering/tcp-filtering.ps>, 2000.

Enhancing NFS Cross-Administrative Domain Access

Joseph Spadavecchia and Erez Zadok

Stony Brook University

{joseph, ezk}@cs.sunysb.edu

Abstract

The access model of exporting NFS volumes to clients suffers from two problems. First, the server depends on the client to specify the user credentials to use and has no flexible mechanism to map or restrict the credentials given by the client. Second, when the server exports a volume, there is no mechanism to ensure that users accessing the server are only able to access their own files.

We address these problems by a combination of two solutions. First, *range-mapping* allows the NFS server to restrict and flexibly map the credentials set by the client. Second, *file-cloaking* allows the server to control the data a client is able to view or access, beyond normal Unix semantics. Our design is compatible with all versions of NFS. We have implemented this work in Linux and made changes only to the NFS server code; client-side NFS and the NFS protocol remain unchanged. Our evaluation shows a minimal average performance overhead and, in some cases, an end-to-end performance improvement.

1 Introduction

NFS was originally designed for use with LANs [17, 22], where a single administrative entity was assumed to control all of the hosts in that site and create unique user accounts and groups. The access model chosen for exporting NFS volumes was simple but weak. In a different administrative domain, the password database may define different users with the same UIDs; a UID clash could occur if files in one domain are accessed from another. Worse, users with local root access on their desktops or laptops can easily access files owned by any other user via NFS, by simply changing their effective UID (i.e., using `/bin/su`).

Therefore, NFS servers rarely export volumes outside their administrative domain. Moreover, administrators resist opening up access even to hosts within the domain, if those hosts cannot be controlled fully. Today, users and administrators must compromise in one of two ways. Either volumes are exported across administrative domains and security is compromised, or the volumes are not exported across administrative domains, preventing users from accessing their data. Neither solution is acceptable.

Although NFSv4 [19] promises to provide strong authentication and provides a convenient framework for

fixing these problems, it will not be available for many platforms and in wide use for several years. The transition between NFSv2 [22] and NFSv3 [2] took around 10 years and corresponds to relatively small changes compared to the changes between NFSv3 and NFSv4. Even today, NFSv3 is not fully implemented on all platforms. Moreover, the NFSv4 specification does not address all of the problems that we wish to fix. Nevertheless, the techniques described in this paper can enhance NFSv4 functionality. For example, whereas NFSv4 optionally supports ACLs (Access Control Lists), it does not specify how to use them to hide files or consider the idea of hiding files.

Current NFS servers implement a simple form of security check for the super user, intended to stop a root user on a client host from easily accessing any file on the exported NFS volume. However, current NFS servers do not allow the restriction and mapping of any number of client credentials to the corresponding server credentials.

We present a combination of two techniques that together increase both security and convenience: *range-mapping* and *file-cloaking*. *Range-Mapping* allows an NFS server to map any incoming UIDs or GIDs from any client to the server's own known UIDs and GIDs. This allows each site to continue to control their own user and group name-spaces separately while allowing users on one administrative domain to access their files more conveniently from another domain. *Range-mapping* is a superset of the usual UID-0 mapping and Linux's *all-squash* option which maps all UIDs or GIDs to -2.

Our second technique, *file-cloaking*, lets the server determine which ranges of UIDs or GIDs should a client be allowed to view or access. We define *visibility* as the ability of an NFS server to make some files visible under certain conditions. We define *accessibility* as the NFS server's ability to permit some files to be read, written, or executed. Cloaking extends normal Unix file permission checks by restricting the visibility and accessibility of users' files when those files are exported via NFS. Cloaking can be used to enforce the NFS client options `nosuid` and `nosgid` which prevent the execution of set-bit files.

Range-mapping and *cloaking* complement each other. Together, they allow NFS servers to extend access to more clients without compromising the existing security

of those files. Whereas ACLs can allow a greater degree of flexibility than cloaking, ACLs are not available on all hosts and all file systems, are not supported in NFSv2, and are partially implemented in NFSv3. Furthermore, ACLs are often implemented in incompatible ways; this is one reason why the new NFSv4 protocol specification lists ACL attributes as optional [19].

Our system is implemented in the Linux kernel-mode NFS server. No changes were made to the NFS client side and our system is compatible with existing NFS clients. This has the benefit that we can deploy our system fairly easily by changing only NFS servers.

We performed a series of general-purpose benchmarks and micro-benchmarks. Range-mapping has an overhead of at most 0.6%. File-cloaking overheads range from 72% for a large test involving 1000 cloaked users—to an improvement of 26% in performance under certain conditions, reflecting a 4.7 factor reduction in network I/O.

The rest of this paper is organized as follows. Section 2 describes the design of our system and includes several examples. We discuss interesting implementation aspects in Section 3. Section 4 describes the evaluation of our system. We review related works in Section 5 and conclude in Section 6.

2 Design

Range-mapping and cloaking are features that offer additional access-control mechanisms for exporting NFS volumes. We designed these features with three goals: compatibility, flexibility, and performance.

First, we are compatible with all NFS clients by requiring no client-side or protocol changes. Range-mapping and cloaking are performed entirely by the NFS server. The server forces a behavior on the client that maintains compatibility with standard Unix semantics. Second, we provide additional flexible access-control mechanisms that allow both users and administrators to control who can view or access files. We allow administrators to mix standard Unix and cloaking semantics to define new and useful policies. Third, our design is economical and efficient. We utilize hash tables and caches to ensure good overall performance.

2.1 Range-Mapping

Traditionally, NFS supports two simple forms of credential mapping called *root-squashing* and *all-squashing*. Root-squashing allows an NFS server to map any incoming UID 0 or GID 0 to another number that does not have superuser privileges, often -2 (the *nobody* user). All-squashing is a Linux NFS server feature that allows the server to map all incoming UIDs or GIDs to another

number, representing a user or group with minimal privileges (e.g., -2).

These two crude forms of credential mapping are not sufficient in many situations. First, there is no way to map arbitrary ranges of client IDs to server IDs. Therefore, squashing is not sufficient in cases where it is desirable to map to more than a single ID. Second, it is useful in some cases to map one set of IDs while squashing all others. For example, if the file system was exported to a client where only UIDs 400–500 should be able to access their files on the server, then it is desirable to assure that all other UIDs coming from the client are squashed to -2.

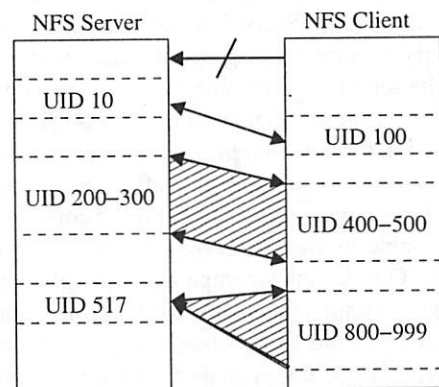


Figure 1: Range-mapping: client UID 100 to server UID 10 (1-to-1); client UIDs 400–500 to server UIDs 200–300 (N-to-N); client UIDs 800–999 to server UID 517 (N-to-1). Note that server UID 517 is reverse-mapped to client UID 800. All other UIDs are restricted.

Range-mapping allows greater flexibility in mapping credentials. Range-mappings are defined per *export* (exported file system) and allow the NFS server to restrict or map any set of an NFS client's credentials to the corresponding set on the server. Figure 1 shows an example of several range-mappings. First, client UID 100 is mapped to server UID 10 (1-to-1). Second, client UIDs 400–500 are mapped to server UIDs 200–300 (N-to-N), respectively. Third, client UIDs 800–999 are mapped to server UID 517 (N-to-1); this case is also called *squashing*. For N-to-1 mappings, the server reverse-maps the single server UID to the first corresponding client UID in the squashed range: server UID 517 is reverse-mapped to client UID 800 in Figure 1. Finally, the arrow with a slash through it means that all other UIDs are restricted (squashed to -2).

Range-mapping is done bidirectionally. Forward mapping is done when a client sends a request to the NFS server and the server maps the user's client UID and GID to the corresponding server UID and GID. Reverse mapping is done when the server responds to the client (i.e., when returning file attributes) and must map the user's

server UID and GID back to the corresponding client UID and GID.

2.1.1 Range-Mapping Configuration Examples

Range-mapping definitions are specified per export in `/etc/exports` as part of the option list. To provide human-readable formatting of the range mapping definitions, we extended the file's format to support whitespace and line continuation. The syntax for the range mapping option is as follows:

```
range_map = rmap_def [rmap_def...]
rmap_def := <uid|gid> rm-low [rm-high]
            <map|squash> lc-low
```

The first option in `rmap_def` specifies whether the definition applies to UIDs or GIDs. The `rm-low`, `rm-high`, and `lc-low` values specify the remote lower bound, remote upper bound, and local lower bound ID values, respectively. The client ID range `rm-low` to `rm-high` is mapped to the server's ID range `lc-low` as follows:

$$lc-low + (rm-high - rm-low)$$

A one-to-one mapping from ID `rm-low` on the client to ID `lc-low` on the server is performed if `rm-high` is not specified or is equal to `rm-low`.

The `map` or `squash` option specifies that an N-to-N, or N-to-1 ID mapping is being performed, respectively. The following example shows a single range mapping definition:

```
/home *.example.com(rw, \
    range_map = \
    uid 100 250 map 12314 \
    gid 100 200 squash 6000)
```

The definition above specifies two range-maps for clients who are members of the `example.com` domain and are accessing the `/home` volume. First, client UIDs 100–250 will be mapped to server UIDs 12314–12464. Second, client GIDs 100–200 be squashed to server GID 6000.

Range-mapping is a superset of root-squashing and all-squashing. Root squashing is a feature of NFS that allows the NFS server to map the root UID (0) and GID (0) to the nobody UID (typically -2) and nobody GID (typically -2). The following example shows how to perform root squashing using range-mapping:

```
/home *.example.com(rw, \
    range_map = \
    uid 0 squash -2 \
    gid 0 squash -2)
```

All-squashing is a feature supported in Linux that maps all client UIDs and GIDs to -2. All-squashing may be done with range-mapping as follows:

```
/home *.example.com(rw, \
    range_map = \
    uid 0 -1 squash -2 \
    gid 0 -1 squash -2)
```

2.2 File-Cloaking

File-cloaking is a mechanism that abstracts the concept of file permissions to allow data to be hidden. With file-cloaking, if a file is not visible then it is not accessible. When a request is made to access a file or list a directory, file-cloaking uses the NFS credentials, file protection bits, and cloaking-mask to determine access and visibility.

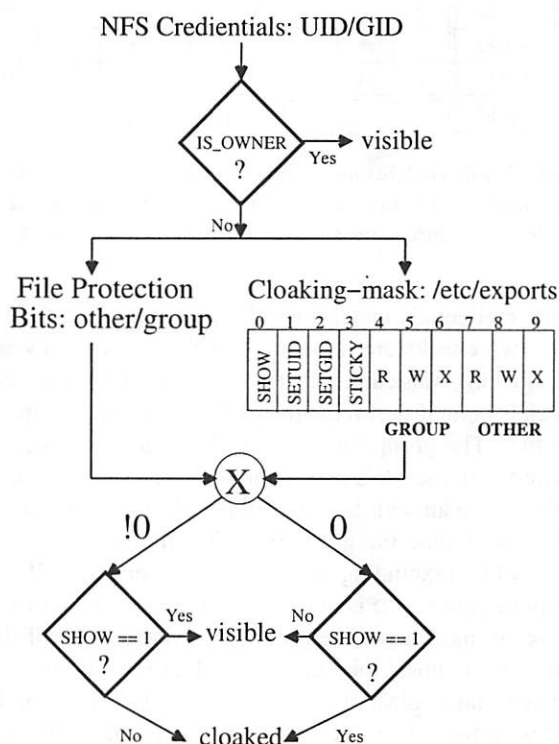


Figure 2: File-cloaking algorithm running on the NFS server.

File	Permission	User	Group
J1	0600	joe	src
J2	0640	joe	src
J3	2666	joe	src
J4	0700	joe	src
E5	0750	ezk	src
E6	0750	ezk	fac
E7	4775	ezk	src
E8	0775	ezk	fac
E9	6700	ezk	src
E10	0000	ezk	src

Table 1: An example listing of files. User `joe` belongs to group `src`, and user `ezk` belongs to groups `src` and `fac`. The leftmost 3 bits attached to the permission mask represent the `SETUID`, `SETGID`, and sticky bits, respectively.)

Figure 2 shows how file-cloaking works. The NFS credentials are checked against the owner of the file.

Cloak Mask	User ezk				User joe						Meaning for files J1–E10
	J1	J2	J3	J4	E5	E6	E7	E8	E9	E10	
+000											Show files to owners only
+007			A				A	A			Show files to owners and others
+070		A	A		A		A	A			Show files to owners and group members
+077		A	A		A		A	A			Show files to all people with any access
−007	v	v		v	v	v			v		Hide world-accessible files
−070	v			v					v		Hide from group members
−077	v			v							Hide from groups members and others
−004	v	A		v	A	v	A	A	v	v	Hide world readable files
−400	v	A	A	v	A	v		A		v	Hide SETUID files
−200	v	A		v	A	v	A	A		v	Hide SETGID files
−000	v	A	A	v	A	v	A	A	v	v	Unix standard

Table 2: File visibility and accessibility for various cloak masks, given the files in Table 1. A “+” in front of the mask implies that files are cloaked by default; a “−” means that files are visible by default. The letter “A” means that the file is visible and accessible. The letter “v” means the file is visible but not accessible. A blank cell means that the file is neither visible nor accessible.

If the credentials match then the file is visible. Otherwise two checks are done, each with an AND between the cloaking-mask and the file’s protection bits. The first check is done if the credentials have group ownership on the file. The group RWX bits of the cloaking-mask are ANDed with the file’s protection bits. The second check is done with an AND between all the bits in the cloaking-mask (excluding the group rwx bits) and the file’s protection bits (excluding the group and user bits). If the result of either AND is non-zero and the SHOW bit of the cloaking-mask is set then the file is visible; else if the SHOW bit is unset then the file is cloaked. The inverse behavior takes place if the result of the AND is zero. In this way, the SHOW bit can determine whether files are visible or cloaked by default, allowing the other mask bits to reverse the default behavior.

Tables 1 and 2 illustrate the concept of cloaking and provide some examples of useful cloaking configurations. Table 1 shows a sample of files owned by two different users: joe and ezk. The permissions for each file are listed including the SETUID, SETGID, and sticky bits. User joe belongs to the src group, and user ezk belongs to the src and fac groups. Table 2 shows a visibility/accessibility matrix of the files listed in Table 1 for eleven useful cloaking masks. In Table 2, “A” implies “v” and the “+” or “−” preceding the mask sets or clears the mask’s SHOW bit, respectively. Next, we describe a few of the examples in Table 2.

With mask +000, the administrator enforces a strict policy that users can only see their own files. On the other hand, with mask −077 the administrator chooses a policy that allows users to decide if their files should be hidden or not by setting the group and world bits. To hide a file from everyone, the user may set any of the “other” bits. To hide a file from group members, the user may set

one of the group bits. Note that this mask is non-intuitive because it restricts access by performing an action that would normally increase others’ access. Also, note that this option is not fail-safe. If the administrator changes the export option hidden files might not only be visible, but also accessible.

The next to last two lines in Table 2 show that cloaking allows the server to enforce the client-side nosuid and nosgid mount options. The nosuid and nosgid options are achieved by cloaking using the masks −400 and −200, which translate to hide SETUID and hide SETGID files, respectively. Note that cloaking here is more restrictive than the nosuid and nosgid options. These options allow files to be executed, but without their set-bits. Cloaking hides such files and thus disallows their execution altogether.

Another example is the mask −002, which hides all world-writable files. This mask can be especially useful on multi-user systems to protect layman users from leaving their files world writable. In this scenario the administrator enforces a policy on the systems users.

2.2.1 File-Cloaking Configuration Examples

Cloaking definitions are specified with range-mappings in the /etc/exports file. The syntax for cloaking definitions is as follows:

```
cloak_list = clist_def [clist_def...]
clist_def := <uid|gid> mask lc-low [lc-high]
```

As with range-mapping, the first option specifies whether the definition applies to UIDs or GIDs. Next, mask is a 10-bit field for defining the cloaking policy. Figure 2 shows the different bits comprising the mask.

The range lc-low to lc-high is the range of server IDs to cloak based on the policy given by the mask. If lc-high

is omitted, then the definition applies only to the server ID *lc-low*. The following example shows a cloaking definition:

```
/home *.example.com(rw, \
    cloak_list = \
    uid +000 500 1000 \
    gid +077 100 200)
```

In this example, there are two cloaking definitions. The first places the restriction for UIDs 500–1000 that only the owners may see or access their files. That is, files on the server owned by UIDs 500–1000 cannot be seen or accessed by any user other than the owner. The second definition states that files owned by UIDs 100–200 are only visible if there is world access to them, or if there is group access to them and the user listing the file belongs to the group of the file.

3 Implementation

We implemented this project in two places: the NFS Utilities (*nfs-utils*) package version 0.3.1 [8] and the NFS server code (both NFSv2 and NFSv3) in the Linux 2.4.4 kernel. We added 1678 lines of code to *nfs-utils*, an increase of 5.8% to its size. This code primarily handles parsing */etc/exports* files for range-mapping and cloaking entries, packing them into exports structures, and passing these structures to the kernel using a special-purpose *ioctl* used by the in-kernel NFS server.

Most of the code we added was to the kernel: 1330 lines of additional code, or an increase of 15.1% to the total size of the NFS server sources. Although this increase is substantial, the bulk of our changes to the kernel code are in new C source and header files, and in stand-alone functions we added to existing source files. The placement of our changes in the kernel sources made it easier to develop: two first-year graduate students spent a combined total of 12 man-weeks developing and testing the code.

3.1 Range-Mapping

For range-mapping we faced three questions: where to do forward mapping, where to do reverse mapping, and how to get the mapping context from the export structures for each client request.

Forward mapping is done in the *nfsd_setuser* function, which is passed a pointer to the relevant export structure; the latter contains the information we need to perform the mapping. Implementing reverse mapping was more difficult. The best place for it was in the server's outgoing path, where it encodes file attributes into XDR structures before shipping them back to the

NFS client. This is done in the *encode_fattr3* routine (or *encode_fattr2* for NFSv2). We find a response packet inside the request structure passed to this function; from this we get the NFS file handle. The latter contains the export information we need to compute the range-mapping.

3.2 Cloaking

Cloaking was more challenging to implement than range-mapping because of the restriction that we only modify the server. Cloaking needs to display different directory listings to each user on the same client. Since clients cache directory contents and file attributes, we have to force the NFS clients to ignore cached information (if any) and reissue an *NFS_READDIR* procedure every time users list a directory. We investigated two options: (1) lock the directory, and (2) fool the client into thinking that the directory's contents changed and thus must be re-read. We chose the second option because locking the directory permanently would have serialized all access to that directory and prevented more than one NFS client from making changes to that directory (such as adding a new file).

To force the client to re-read directories, we increment the in-memory modification time (*mtime*) of the directory each time it is listed; we do not change the actual directory's *mtime* on disk. This technique has been used before to prevent client-side caching in NFS-based user level servers [14, 25, 29]. NFS clients check the *mtime* of remote directories before using locally cached data. Since the *mtime* always changes, the clients re-read the directory each time and effectively discard their local cache. The *mtime* field has a resolution of one second, but sometimes several *readdir* requests come in one second. We therefore had to ensure that the *mtime* is always incremented on each listing. This has a side effect that the modification time of directories being listed frequently could move into the future. In practice this was not a problem because directory-reading requests are often bursty and in between bursts the real clock has a chance to catch up to a directory's *mtime* that may be in the near future. Furthermore, future Linux kernels will increase the *mtime* resolution to microseconds, thus practically eliminating this problem.

We expected that forcing the clients to ignore their directory caches will reduce performance. However, if a client machine has only one user (as is the case with most personal workstation and laptops), we can allow the client to cache directory entries normally since there is little risk that another user on that client will be able to view cached entries. We made client caching optional by adding a server-side export option called *no_client_cache* that, if enabled, forces the direc-

tory mtime to increase and cause clients not to cache directory entries. If `no_client_cache` is not used (the default), we do not increase the mtime and NFS clients cache directory entries normally.

Cloaking requires that some files be hidden from users and therefore those files' names should not be sent back to the NFS client. We implemented this in the `encode_entry` function. Given a file's owner, group, mode bits, and the export data structures, we compute whether the file should be visible or not. If the file is invisible, we simply skip the XDR encoding of that file. If the file is not invisible, then we allow access to that file based on normal Unix file permissions. A user could try to lookup (perhaps guess) a file that is hidden to that user. To catch this we perform a cloaking check also in `nfsd_lookup` and if the file should be invisible to the calling user, we return an error code back to the lookup request.

4 Evaluation

To evaluate range-mapping and cloaking in a real world operating environment, we conducted extensive measurements in Linux comparing vanilla NFS systems against those with different configurations of range-mapping and cloaking. We implemented range-mapping and file-cloaking in NFSv2 and NFSv3; however, we report the results for NFSv3 only. Our benchmarks for NFSv2 show comparable performance. In this section we discuss the experiments we performed with these configurations to (1) show overall performance on general-purpose workloads, and (2) determine the performance of individual common file operations that are affected the most by this work. Section 4.1 describes the testbed and our experimental setup. Section 4.2 describes the file system workloads we used for our measurements. Sections 4.3 and 4.4 present our experimental results.

4.1 Experimental Setup

We ran our experiments between an unmodified NFS client and an NFS server using five different configurations:

1. **VAN:** A vanilla setup using an unmodified NFS server. Results from this test gave us the basis on which to evaluate the overheads of using our system.
2. **MNU:** Our modified NFS server with all of the range-mapping and cloaking code included but not used. This test shows the overhead of including our code in the NFS server while not using those features.

3. **RMAP:** Our modified NFS server with range-mapping configured in `/etc/exports`. Since our range-mapping code works exactly the same when a single UID or a range of UIDs are mapped, for simplicity these tests mapped a single UID.
4. **CLK:** Our modified NFS server with cloaking configured in `/etc/exports`. To illustrate the worst-case performance for cloaking, we set the cloaking mask to `+000`, indicating the most restrictive cloaking possible. Using `+000` ensures that the code which determines if a file should be visible or not checks as many mask conditions as possible, as we described in Section 2.
5. **RMAPCLK:** Our modified NFS server with both range-mapping and cloaking configured in `/etc/exports`. To ensure worst-case performance, we set the user entries that are range-mapped so they are also cloaked.

The last three configurations were intended to show the different overheads of our code when each feature is used alone or combined. Since our system runs the same range-mapping or cloaking code with either UIDs or GIDs we evaluated range-mapping and cloaking only for UIDs.

All experiments were conducted between two equivalent Dell OptiPlex model GX110 machines that use a 667MHz Intel Pentium III CPU, 192MB of RAM, and a Maxtor 30768H1 7.5GB IDE disk drive. The two machines were connected to a stand-alone dedicated switched 100Mbps network.

To ensure that our machines were equivalent and our setup was stable, we ran the large-compile Am-utils benchmark (see Section 4.2.1) on both machines, alternating the server and client roles of the two. We compiled the package on the client, using an exported file system from the server. We compared the results and found the difference in elapsed times to be 1.003% and the difference in system time (as measured on the client) to be 1.012%. The standard deviations for these tests ranged from 2.2–2.7% of the mean. Therefore, for the purposes of evaluating our systems, we consider these machines equivalent, but we assume that benchmark differences smaller than 1% may not be significant.

We installed a vanilla Linux 2.4.4 kernel on both machines. We designated one machine as client and installed on this machine an unmodified NFS client-side file system module. On the server we installed both a vanilla NFS server-side file system module and our modified NFS server module that included all of the range-mapping and cloaking code. Since our code exists entirely in the NFS server, we could use a vanilla kernel on the server and simply load and unload the right `kNFSd` module.

On the server we installed our modified user-level NFS utilities that understand range-mapping and cloaking; these include the `exportfs` utility and the `rpc.nfsd`, `rpc.lockd`, and `rpc.mountd` daemons. Finally, the server was configured with a dedicated 334MB EXT2FS partition that we used exclusively for exporting to the client machine.

All tests were run with a cold cache on an otherwise quiescent system (no user activity, periodic `cron(8)` jobs turned off, unnecessary services disabled, etc.). To ensure that we used a cold cache for each test, we unmounted all file systems that participated in the given test after the test completed, and we mounted the file systems again before running the next iteration of the test (including the dedicated server-side exported EXT2FS partition). We also unloaded all NFS-related modules on both client and server before beginning a new test cycle. We verified that unmounting a file system and unloading its module indeed flushes and discards all possible cached information about that file system.

We ran each experiment 20 times and measured the average elapsed, system, and user times. In file system and kernel benchmarks, system times often provide the most accurate representation of behavior. In our tests we measured the times on the NFS client, but the code that was modified ran on the server's kernel. System times reported on a client do not include the CPU time spent by the server's kernel because when the server is working on behalf of the client, the client's user process is blocked waiting for network I/O to complete. I/O wait times are better captured in the elapsed time measurements (or, alternatively, when subtracting system and user times from elapsed times). Since our testbed was dedicated and the network connection fast, we chose to report elapsed times as the better representatives of the actual effort performed by both the client and the server.

Finally, we measured the standard deviations in our experiments and found them to be small: less than 3% for most benchmarks described. We report deviations that exceeded 3% with their relevant benchmarks.

4.2 File System Benchmarks

We measured the performance of our system on a variety of file system workloads and with the five different configurations as described in Section 4.1. For our workloads, we used three file system benchmarks: two general-purpose benchmarks for measuring overall file system performance and one micro-benchmark for measuring the performance of common file operations that may be affected by our system.

4.2.1 General-Purpose Benchmarks

Am-utils The first general-purpose benchmark we used to measure overall file system performance was `am-utils` (The Berkeley Automounter) [13]. This benchmark configures and compiles the `am-utils` software package inside a given directory. We used `am-utils-6.0.7`: it contains over 50,000 lines of C code in 425 files. The build process begins by running several hundred small configuration tests intended to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 265 additional files. Overall, this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as `unlink`, `mkdir`, and `symlink`. The main usefulness of this benchmark is to show what the overall performance might be for regular use of our system by users.

NFSSTONE The second general-purpose benchmark we used was an NFS-specific benchmark called `NFSSTONE` [18]. This traditional benchmark performs a series of 45522 file system operations, mostly executing system calls, to measure how many operations per second can an NFS server sustain. The benchmark performs a mix of operations intended to show typical NFS access patterns [16]: 53% LOOKUPS, 32% READS, 7.5% READLINKS (symlink traversal), 2.3% GETATTRS, 3.2% WRITES, and 1.4% CREATES. This benchmark performs these operations as fast as it can and then reports the average number of operations performed per second, or `NFSSTONES`.

For these two general benchmarks, we set up the NFS server with the `/etc/exports` file configured as follows:

- For the `VAN` and `MNU` tests, the exports files contained only one entry exporting a single file system. No range-mapping or cloaking were configured.
- For the `RMAP` test we configured 10 range-mapped users, representing a small configuration we believe will be common. The user we ran these benchmarks on the client was one of the mapped UIDs. This way we caused the server to do some work, bidirectionally mapping one user ID to another.
- For the `CLK` test we configured 10 cloaked users. The user we ran the benchmarks on the client was one that was allowed to access and modify their files on the server. This test shows the worst-case scenario for cloaking, when the server has to look at the entire list of cloaked users and not find the user who is accessing the files.
- For the `RMAPCLK` test we configured an `/etc/exports` file that contained 10 range-mapped entries and also 10 cloaked entries. The

user we ran the test on the client was range-mapped and had permission to view and modify their files.

We investigated three additional benchmarks that we did not use to evaluate our work. First, the Modified Andrew Benchmark (MAB) [11] is also a compile-based benchmark but it is too small for modern hardware and completes too quickly as compared to the larger *am-utils* compile. Second, a newer version of NFSSTONE called *NHFSSTONE* [7] uses direct RPC calls to a remote NFS server instead of executing system calls on the client because the latter can result in a different mix of actual NFS server operations that are executed. Unfortunately, the only available *NHFSSTONE* benchmark for Linux [8] supports only NFSv2, whereas we wanted a benchmark that could run on both NFSv2 and NFSv3 servers [2, 12, 17, 22]. Third, the SFS 2.0 benchmark, a successor to LADDIS [24], is a commercial benchmark that provides an industry-standardized performance evaluation of NFS servers (both NFSv2 and NFSv3), but we did not have access to this benchmark [14, 23]. Nevertheless, we believe that the benchmarks we did perform represent the performance of our system accurately.

4.2.2 Micro-Benchmarks

GETATTR: The third benchmark we ran is the primary micro-benchmark we used. Our code affects only file system operations involving accessing and using file attributes such as owner, group, and protection bits. This benchmark runs a repeated set of recursive listing of directories using `ls -lR`. That way, the micro-benchmark focuses on the operations that are affected the most: getting file attributes and listing them.

Since this benchmark is the one that is affected the most by our code, we ran this test repeatedly with several different configurations aimed at evaluating the performance and scalability of our system. To ensure that the server had the same amount of disk I/O to perform, we used fixed-size directories containing exactly 1000 files each.

To test the scalability, we ran some tests with a different numbers of range-mapped or cloaked entries in `/etc/exports`: 10, 100, and 1000. Ten entries intends to represent a small site whereas one-thousand entries represents a large site.

For the *VAN* and *MNU* tests we used a directory with 1000 zero-length files owned by 1000 different users. These two tests show us the base performance and the effect on performance that including our code has, respectively.

For the *RMAP* test we also kept the size of the directories being listed constant, but varied the number of UIDs being mapped: 10 mapped users each owning 100 files, 100 mapped users each owning 10 files, and 1000 users

each owning one file. The directories were created such that each user's files were listed together so we could also exercise our range-mapping UID cache. The user that ran the `ls -lR` command for this benchmark on the client was one of the mapped UIDs. (It does not matter which of the mapped users was the one running the test since the entire directory was listed and for each file the server had to check if range-mapping was applicable.) These tests show what effect range-mapping has on performance for different scales of mapping.

For the *CLK* test we used a similar setup as with range mapping: directories containing 1000 files owned by a different number of cloaked users each time: 10, 100, and 1000. To make the server perform the most work, we ran the benchmark on the client using a user whose UID was not permitted to view any of the files listed. This ensured that the server would process every file in the directory against the user who is trying to list that directory, but would not return any file entries back to the client. This means that although the directory contains 1000 files on the server, the client sees empty directories. This has the effect of reducing network bandwidth and the amount of processing required on the client.

The *RMAPCLK* test combined the previous two tests, using a different number of range-mapped users all of whom were cloaked: 10, 100, and 1000. The directories included 1000 files owned by a corresponding number of users as were mapped and cloaked. The user we ran the test as, on the client, was one of those users to ensure that the server had to process that user's UID both for mapping and cloaking. This guaranteed that only the files owned by the cloaked user (out of a total of 1000 files) would be returned to the client: 100 files were returned when there were 10 cloaked users, 10 files returned when there were cloaked 100 users, and only one file returned when there were 1000 cloaked users. This test therefore shows the combination of two effects: range-mapping and cloaking make the server work harder, but cloaking also results in reducing the number of files returned to the client, and thus saving on network bandwidth and client-side processing.

Finally, for all benchmarks involving cloaking (*CLK* and *RMAPCLK*) we ran the tests with and without the `no_client_cache` option (Section 3.2), to evaluate the effects of circumventing NFS client-side caching.

4.3 General-Purpose Benchmark Results

Am-Utils Figure 3 shows the results of our *am-utils* benchmark. This benchmark exercises a wide variety of file operations, but the operations affected the most involve getting the status of files (via `stat(2)`), something that does not happen frequently during a large compilation; more common are file reads and writes. There-

fore the effects of our code on the server are not great in this benchmark, as can be seen from the individual results.

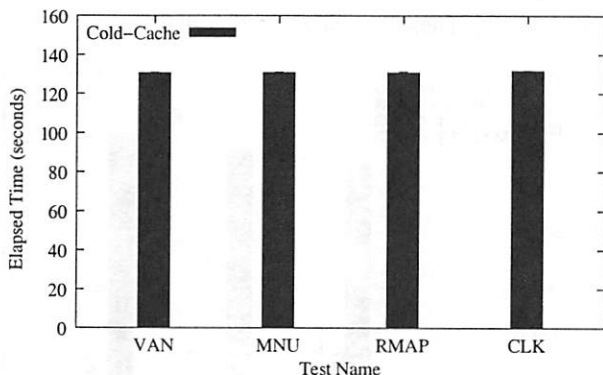


Figure 3: Results of the *am-utils* benchmark show a small difference between all of the tests—less than 1% between the fastest and slowest results.

When our code is included but not used (MNU vs. VAN) we see a 0.14% degradation in performance. Adding range-mapping (RMAP vs. MNU) costs an additional 0.026% in performance. Adding cloaking (CLK vs. MNU) costs an additional 0.54% in performance. These results suggest that range-mapping and cloaking have a small effect on normal use of NFS mounted file systems.

NFSSTONE Figure 4 shows the results of the NFSSTONE benchmark. This benchmark exercises the operations that would be affected by our code on the server (GETATTR) more times than the Am-utils benchmark. Therefore, we see higher performance differences between the individual tests.

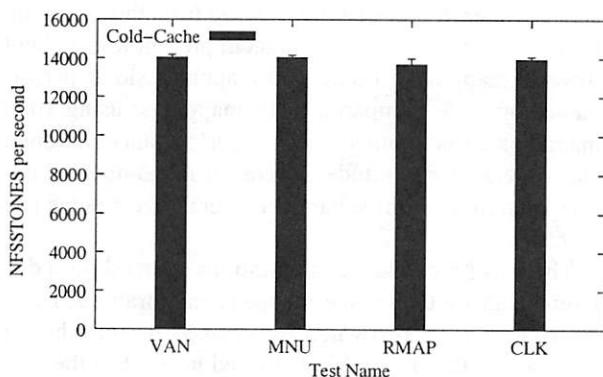


Figure 4: Results of the *NFSSTONE* benchmark show a small difference between all of the tests—less than 2.6% between the fastest and slowest.

When our code is included but not used (MNU vs. VAN) we see a small 0.05% degradation in performance, suggesting that when our code is not used, it has little effect on performance. Adding range-mapping (RMAP vs.

MNU) costs an additional 2.5% in performance. Adding cloaking (CLK vs. MNU) costs an additional 0.54% in performance. These results also suggest that range-mapping and cloaking have a small effect on normal use of NFS mounted file systems.

4.4 Micro-Benchmark Results

Our code affects operations that get file attributes (*stat(2)*). The micro-benchmarks tested the effect that our code has on those operations using a variety of server configurations.

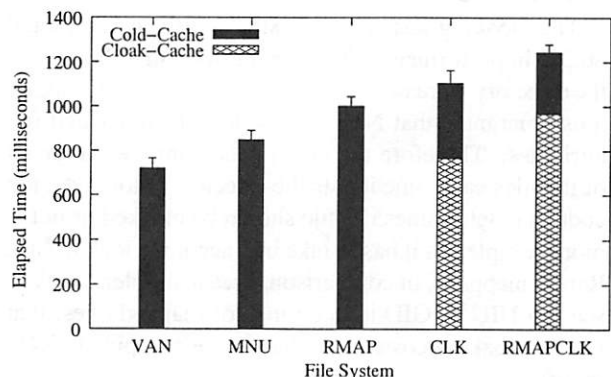


Figure 5: Results of recursive listing of directories containing exactly 1000 entries on the server. The tests were configured to result in exactly 1000 files being returned to the client each time (i.e., worst case). The Cloak-Cache bars show results when NFS clients used cached contents of cloaked directories (i.e., the *no_client_cache* export option was not used). Standard deviations for this tests were 4.3–6.5% of the mean.

Figure 5 shows results of our first micro-benchmark: a recursive listing (*ls -lR*) we conducted on an NFS-mounted directory containing 1000 entries. The */etc/exports* file on the server was configured with range-mapping and cloaking such that the user listing that directory on the client would always see all files. This ensures that the amount of client-side work, network traffic, and server-side disk I/O for accessing the directory remained constant, while making the server's CPU experience different workloads.

The black bars in Figure 5 show the worst-case results, when NFS clients were ignoring their cache: *no_client_cache* was used on the server, described in Section 3.2. The Cloak-Cache bars show the results when this option was not used, thus allowing clients to use their directory caches.

When our code is included but not used (MNU vs. VAN) we see a 17.6% degradation in performance. This is because each time a client asks to get file attributes, our code must check to see if range-mapping or cloaking are configured for this client. This checking is done by comparing four pointers to NULL. Although these checks

are simple, they reside in a critical execution path of the server's code—where file attributes are checked often.

Adding range-mapping (RMAP vs. MNU) costs an additional 18.1% in performance. This time the server scans a linked list of 1000 range-mapping entries for the client, checking to see if the client-side user is mapped or not. Each of 1000 files were owned by a different user and one user was range-mapped. Therefore the server had to scan the entire list of range mappings for each file listed; only one of those files actually got mapped. Such a test ensures that the range-mapping cache we designed was least effective (all cache misses), to show the worst-case overhead.

The cloaking test (CLK vs. MNU) costs an additional 30.3% in performance. This test ensured that the files in the directory were not owned by a cloaked user. Cloaking must guarantee that NFS clients do not use cached file attributes. Therefore the client gets from the server all of the files each time it lists the directory. Moreover, the code that determines if a file should be cloaked or not is more complex as it has to take into account cloak masks. Range-mapping, in comparison, uses a simpler check to see if a UID or GID is in a range of mapped ones; that is why cloaking costs more than range-mapping (10.4% more).

The cumulative overhead of range-mapping and cloaking, when computed as the worst case time difference between cloaking and range-mapping, compared against MNU, is 48.4%. However, when combining cloaking and range-mapping together (RMAPCLK vs. MNU) we see a slightly smaller overhead of 46.9%. This is because the range-mapping and cloaking code is localized in the same NFS server functions. This means that when the NFS server invokes a function look up a file, both range-mapping and cloaking tests are performed without having to invoke that function again, pass arguments to it, etc.

The worst-case situation (RMAPCLK vs. VAN) has an overhead difference of 72.3%. This overhead is for the inclusion of our code and its use on a large directory with 1000 files. Although this overhead is high, it applies only when getting file attributes. Under normal user usage, performance overhead is smaller, as we have seen in Section 4.3.

Finally, for single-user NFS clients, the server can safely allow such hosts to cache directory entries, thus improving performance. The Cloak-Cache results in Figure 5 show what happens when we did not use the `no_client_cache` export option. For the CLK test, client-side caching improves performance by 30.6%. Client-side caching improves performance of the RMAPCLK test by 22.3%.

The micro-benchmarks in Figure 5 show the worst case performance metrics, when both the server and

client have to do as much work as possible. The second set of micro-benchmarks was designed to show the performance of more common situations and how our system scales with the number of range-mapped or cloaked entries used. These are shown in Figure 6.

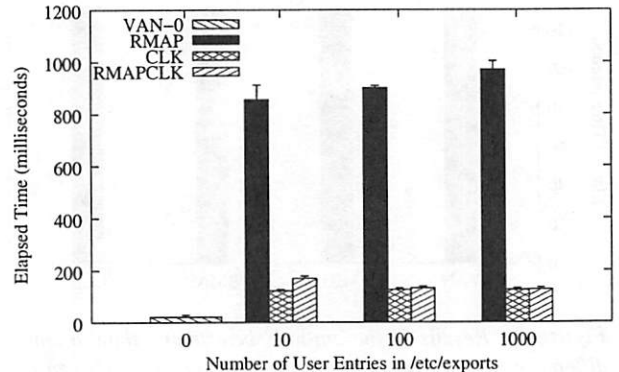


Figure 6: Results of recursive listing of directories containing a different number of files while the server is configured with a different number of mapped or cloaked entries. Standard deviations for this tests were 4.3–6.7% of the mean.

The range-mapping bars (RMAP) show the performance of listing directories with 1000 files in them, but varying the number of users that were range-mapped. Range-mapping with 10 users implies that each user owns 100 files. When one of those users lists the directory, that user sees 1000 files, but 100 of those files have their UID mapped by the server. For 100 files and users, only 10 files are mapped; for 1000 files and users, only one file is mapped. The largest cost for range-mapping is the number of range-map entries listed in `/etc/exports` because the server has to scan the entire (in-memory) list to determine the range-mapping needed for each user that is accessing from the client and for each file that the server wants to present to the client (reverse mapping). Using 100 mappings slows performance by 5.3% compared to 10 mappings; using 1000 mappings costs another 7.9% in performance. Despite two orders of magnitude difference in number of entries used in these three bars, the overall overhead is just 13.6%.

The bars for cloaked configurations (CLK) show a different behavior than range-mapped configurations. Here, all 1000 files were owned by cloaked users, whether there were 10, 100, or 1000 cloaked users. But the user that listed the files on the client was not one of those cloaked users and therefore was not able to see any of those files; what they listed appeared on the client as an empty directory. This test fixes the amount of work that the client has to do (list an empty directory) and the server's work (scan 1000 files and apply cloak rules to each file). This test differs in the number of cloaked user entries listed in `/etc/exports`. The bars show a

small difference in the amount of work that the server had to do to process the cloak lists: 4.1% performance difference between the largest and smallest lists. However, just using the cloaking code costs in performance, even if the client receives no files. We compared this cloaking to listing an empty directory on a vanilla NFS server without cloaking code (marked as VAN-0 in Figure 6); cloaking is 17.8 times slower than listing an empty directory, showing that whereas the client has little work to do, the server must still process a large directory.

Although cloaking consumes more CPU than range-mapping (as seen in Figure 5), the difference between the bars is smaller than with range-mapping and the bars themselves are smaller: 7.3 times faster. The main reason for this is that this cloaking test returns no files back to the client, saving a lot on network bandwidth and client-side CPU processing. This shows on one hand an interesting side effect to cloaking: a reduction in network I/O and client-side processing. On the other hand, to make cloaking work reliably, we had to ensure that the clients would not use cached contents of directories with cloaked files. This potentially increases the amount of network I/O and client-side processing as clients have to scan directories through the server each time they list a directory. We explore these opposing interactions next.

The last set of bars in Figure 6 shows the performance when combining range-mapping with cloaking (RMAP-CLK). Since all of the users' files were cloaked and range-mapped, and the user that listed the directory on the client was one of those users, then that user saw a portion of those files (the files they own). With 10 cloaked users, 100 files were seen by the client; with 100 cloaked users, 10 files were seen; and with 1000 cloaked users, only one file was seen. This means that the amount of work performed by the client should decrease as it lists fewer files and has to wait less time for network I/O. The RMAPCLK bars indeed show an *improvement* in performance as the number of cloaked user entries increases. The reason for this improvement is that the savings in network I/O and client-side processing outweigh the increased processing that the server performs on larger cloak lists. Listing the same directory when we use 100 cloaked and range-mapped entries is 22.3% faster than the directory with 10 entries, because we are saving on listing 90 files. Listing the directory with cloaked 1000 entries is only an additional 4% faster because we are saving on listing just 9 files.

To find out how much cloaking saves on network I/O, we computed an estimate of the I/O wait times by subtracting client-side system and user times from elapsed times. We found that for a combination of cloaking and range-mapping with 10 users, network I/O is reduced by a factor of 4.7. Since cloaking with the `no_client_cache` option forces clients to ignore

cached directory entries, these immediate savings in network I/O would be overturned after the fifth listing of that directory. However, without the `no_client_cache` option, network I/O savings will continue to accumulate.

5 Related Work

The closest past related works to ours are BSD-4.4 `umapfs` and the older and now defunct user-level NFS server for Linux, `Unfsd` [5]. BSD-4.4 `umapfs` works on the client and is not enforced by the server; thus credentials cannot be controlled by the server and this solution is not as secure as our server-side range-mapping. The Linux `Unfsd` server included extensions that supported UID and GID mapping or squashing via NIS or an external RPC server called `ugidd`. This user-level NFS server had several serious deficiencies: it was slow due to context switches and data copies between user level and the kernel; it did not reliably support many important features such as the RPC LOCKD protocol; and it had several security flaws that could not be solved unless the server ran in kernel mode. For those reasons current versions of Linux include a kernel-mode NFS server named `kNFSd`. When that server code was written (from scratch), it did not include support for range-mapping. Our work has added support for range-mapping, squashing, and cloaking into the Linux kernel. We achieved good performance while offering flexible features that were not available before (e.g., cloaking), not even in `Unfsd`.

Today's NFS servers include a feature to suppress access from UID 0 or GID 0, also known as *root squashing* [2, 12, 17, 22]. Linux also includes a feature called *all squashing* that maps all incoming UIDs or GIDs to a single number. As we saw in the design section, our work is a superset of these forms of UID or GID squashing (root or all). With file-cloaking, we support a superset of masking features that includes the ability to disable SETUID or SETGID binaries from executing over NFS.

NFS runs on top of RPC (Remote Procedure Calls) and it is possible to secure NFS by securing the RPC layer. Several secure-RPC systems exist that support even strongly-secure systems such as Kerberos [1, 9, 21]. Unfortunately, past secure RPC systems used with NFS did not always interoperate well with each other and are not available for all existing NFS implementations. To use a system such as Kerberos, NFS clients, servers, and even some user applications have to be modified to support Kerberos; consequently, most NFS systems use the weaker form of user authentication known as AUTH_UNIX, where NFS clients inform the servers what UID and GID they use. Even with strong RPC security, NFS servers still do not support features such as cloaking and range-mapping. Our work does not aim

to replace strong security, but rather to show how additional access methods that improve security can be implemented and deployed easily, and that these changes have little effect on overall performance.

A newer version of the protocol, NFSv4, promises to support many new features including mandatory strong security, protocol extensibility, support for wide-area file access, and better interoperability between Unix and non-Unix systems [19,26]. In NFSv4, users need not be determined by their UID and GID, but by a universal identifier such as an Email address or an electronic signature. Doing so will help identify users uniquely throughout the Internet and would alleviate the need for range-mapping.

Although the current NFSv4 specification does not support cloaking, the protocol was designed for extensibility. Our cloaking techniques do not require a change in the NFS protocol and can be implemented in exactly the same way: our work is therefore compatible with NFSv4 as well as with older NFS protocols. However, as discussed in Section 3.2, our cloaking implementation may be configured to force NFS clients not to cache file attributes or use them, to ensure the correctness of the data that is given to different users on the same client. This costs in performance as we saw in Section 4.4. With NFSv4, cloaking could be added easily as extensions to the protocol that cooperative NFSv4 clients and servers can agree on dynamically. An NFSv4 server can allow an NFSv4 client to cache these entries. If another user tries to list a directory that is cached on the server, the server can then issue a *callback* request to the client (now acting as a small server for these RPC callbacks) to flush the cache before the server sends the client a list of files for that directory; this new list can be different based on the particular view of that directory for another user.

NFSv4 also supports Access Control Lists (ACLs) as optional file attributes. ACLs are also possible with NFSv3, but they are not part of the specification and not all vendors implement ACL support. The reason is that not all operating systems and file systems support ACLs, and those that do are often incompatible (for example, one system supports ACLs only for directories and another system supports them for any file). Whereas ACLs can be an effective and powerful access-control mechanism and are compatible with our cloaking techniques, ACL support remains an optional feature of existing and future NFS protocols.

NFSv2 was widely used in 1994, when NFSv3 was introduced. The protocol has not changed fundamentally: two defunct RPC operations were removed and a few more added; support for TCP and 64-bit files was included too. Still, it took several years before most major vendors began supporting NFSv3 and a few more years to stabilize their code. Today, eight years later, not all

vendors who support NFSv2 also support NFSv3, and of those that do, interoperability and stability problems remain. In comparison, NFSv4 represents a large departure from NFSv3: the now stateful protocol integrates all previous RPC services needed to support NFS (such as MOUNTD, LOCKD, and more) and the specification is larger too. The NFSv4 RFC is 212 pages, compared with only 126 pages for the NFSv3 RFC. We expect that it would be several years before NFSv4 is deployed by all vendors and is stable. Even then, older versions of NFS are likely to remain in use. For those reasons, we believe that the work we presented in this paper remains viable for the foreseeable future.

There are several commercial products that can map credentials transparently, such as Network Appliances's filer [10]. Also, AFS [4] can map users transparently by referring to their usernames. However, these solutions are often expensive, are not available as Open Source Software, or not in wide use compared to Linux and NFS.

6 Conclusions

The main contribution of this work is to allow NFS servers to export their file systems to hosts they would not have allowed access before for security reasons. We used two techniques: *range-mapping* and *file-cloaking*.

Range-Mapping translates UIDs and GIDs of users and their files between NFS servers and clients that exist on different administrative domains. This lets users access their files on different sites where their user accounts have different credentials. File-cloaking prevents certain files from being seen or accessed by some users. This lets administrators and users control the accessibility and visibility of their files. One use of this is to allow only the files' owners to see their own files; or to prevent world-writable or SETUID files from being seen or accessed by anyone other than their owners; or ensuring that only members of the same Unix group could see and write to a shared file. Cloaking gives administrators the flexibility to safely export multi-user file systems to clients of a single user where that user may not be trusted with the remaining files on the exported volume.

We designed and implemented this work to be contained entirely inside the NFS server and require no changes to the NFS protocol or clients. This ensured that our work can interoperate with all existing NFS clients without modification.

Special care was taken to ensure that performance of our system was good and that kernel resource consumption remained low. Our benchmarks show a good performance even with a large number of range-mapping and file-cloaking entries in use.

6.1 Future Work

Cloaking is a feature that is useful not just for NFS but for all file systems. We plan on moving our cloaking code to the Virtual File System (VFS) [6, 15]. This way, cloaking features could be used with other native file systems such as EXT2FS on local hosts, or with stackable file systems [3, 20, 27, 28, 30], as well as NFS and other network-based file systems.

We plan to explore methods to improve the performance of cloaking. To improve cloaking performance, we have to allow multi-user NFS clients to cache files. However, the VFS's directory cache is not aware of which users are looking up entries in the cache. If one user lists a directory, all the files in that directory are cached and can be seen by another user that can list the same directory. For cloaking to work with caching, we need to support *user views* of the directory cache. That is, each entry in the client-side cache should have enough information about cloaking to determine how to present the contents of a directory to a user based on their immediate credentials, filtering files that they should not see or access. To achieve this, it would be important to change the directory cache and the VFS in a way that does not change existing NFS protocols.

An alternative to changes to the directory cache is to implement cloaking functionality in both the NFS server and the client. However, changing many existing NFS clients is impractical and changing NFS protocols in use is nearly impossible. A more suitable platform for such changes would be NFSv4 as we discussed in Section 5. NFSv4 standardizes the mechanisms to extend the protocol and provides callback methods for servers to initiate contact with clients. In this way we can allow clients to cache directories that were filtered due to cloaking, and force new client-side users to contact the server to get an updated list of files for a directory, based on a different view of that directory for the new user.

7 Acknowledgments

We thank Nenad Dedic for contributing to the early range-mapping and file-cloaking code, for Charles Wright for his comments while the paper was being written, and for R. Sekar for his suggestion to add a `no_client_cache` export option. Thanks go to the anonymous Usenix reviewers, and to Chris Demetriou for his helpful comments. This work was partially made possible by an HP/Intel gift number 87128.

The work described in this paper is Open Source Software and is available for download from <ftp://ftp.fsl.cs.sunysb.edu/pub/enf>.

References

- [1] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *Proceedings of the Winter USENIX Technical Conference*, pages 253–67, Winter 1991.
- [2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.
- [3] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [4] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [5] O. Kirch. The Linux user-space NFS server. <ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir>, December 1997.
- [6] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.
- [7] Legato Systems, Inc. NHFSSTONE – Network File System benchmark program. <ftp://wuarchive.wustl.edu/languages/c/unix-c/benchmarks/nhfsstone.tar.Z>, July 1989.
- [8] H. J. Lu. Linux NFS utility package. <http://nfs.sourceforge.net>, February 2001.
- [9] S. Lunt. Experiences with Kerberos. In *Proceedings of the Second USENIX Security Workshop*, pages 113–20, August 1990.
- [10] Network Appliance, Inc. NetApp files. <http://www.netapp.com>, 2002.
- [11] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Technical Conference*, pages 247–56, Anaheim, CA, Summer 1990. USENIX.
- [12] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, June 1994.

- [13] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.0.4 edition, February 2000. <http://www.am-utils.org>.
- [14] D. Robinson. The advancement of NFS benchmarking: SFS 2.0. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA '99)*, pages 175–185, Seattle, WA, November 1999.
- [15] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.
- [16] R. Sandberg. The Sun network file system: Design, implementation and experience. Technical report, Sun Microsystems, 1985.
- [17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–30, Summer 1985.
- [18] B. Shein, M. Callahan, and P. Woodbury. NFS-STONE: A network file server performance benchmark. In *Proceedings of the Summer USENIX Technical Conference*, pages 269–275, Baltimore, MD, Summer 1989.
- [19] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3010, Network Working Group, December 2000.
- [20] G. C. Skinner and T. K. Wong. “Stacking” Vnodes: A progress report. In *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.
- [21] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter USENIX Technical Conference*, pages 191–202, Winter 1988.
- [22] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.
- [23] The Standard Performance Evaluation Corporation. SPEC SFS97 (2.0) benchmark. <http://www.spec.org/osg/sfs97>, June 2001.
- [24] A. Watson and B. Nelson. LADDIS: A multi-vendor and vendor-neutral SPEC NFS benchmark. In *Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI)*, pages 17–32, October 1992.
- [25] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.
- [26] E. Zadok. *Linux NFS and Automounter Administration*, chapter 6: NFS Version 4, pages 151–180. Sybex, Inc., May 2001.
- [27] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, May 1999.
- [28] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.
- [29] E. Zadok and A. Dupuy. HLFSD: Delivering Email to your \$HOME. In *Proceedings of the Seventh USENIX Systems Administration Conference (LISA VII)*, pages 243–254, Monterey, CA, November 1993.
- [30] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

Ningai: A Linux Cluster for Business

Andrew Hume

AT&T Labs - Research
andrew@research.att.com

Scott Daniels

Electronic Data Systems Corporation
scott.daniels@eds.com

Abstract

Clusters of commodity PC hardware are very attractive as a basis for large scale computing. In fact, this style of system, commonly referred to as Beowulf systems, are well on their way to dominating the supercomputing arena, which is almost solely concerned with large scientific computing. In other domains, issues other than performance predominate.

This paper describes Ningai, an architecture for computing on partitioned datasets using a cluster of loosely coupled computers. Although performance is a goal, it is dominated by the goals of availability, reliability, scalability and manageability. We describe the architecture, together with a large-scale application, and describe how we met our goals. We also discuss how well our platform fared, and the lessons we have learned. As our environment is jargon rich, we include a glossary.

1. Introduction

The relentless pressure of Moore's Law, or more accurately, the surprisingly continuous exponential growth in CPU speed and disk capacity, enables, and ultimately forces, the development of systems architectures to handle problems of ever increasing size. No where is this trend more obvious than in scientific supercomputing where the fastest systems, traditionally large expensive monolithic machines from Cray, Fujitsu and the like, will soon be vast arrays of PCs linked in architectures popularised as Beowulf systems[Bec95].

The adoption of this kind of architecture for business systems has been rather less enthusiastically pursued. Despite the promise of cheaper hardware and system software costs, issues of reliability, availability and the necessity to run common commercial software packages have dominated and largely blocked the introduction of these architectures. In particular, the commonplace property of scientific Beowulf systems that a computation fails completely if any node fails during the computation, while apparently tolerable for the scientific community, is *not*

viable for business systems.

Over the last year, we have been in a position to investigate these issues in the context of a large production system within AT&T, namely Gecko[Hum00]. Gecko traced the processing of telephone call and billing records for residential AT&T customers on a continuous and ongoing basis. In 2001, we were asked to resurrect Gecko, with the change of focus to analysing business (rather than residential) customers and having rather less funding for the project. We took the liberty of interpreting this request as a desire to port the Gecko software to a cluster of PCs running Linux (or *somesuch*) and adding some requirements of our own:

- the system must be highly available and reliable, requiring only 8x5 support (and not 24x7), even if hardware fails.
- the system must be highly scalable. That is, we can predict with confidence how much CPU and disk resources are needed for a desired workload, and that the resources are roughly linear in the size of the workload.
- running the system, which has two major components (managing the hardware and

managing the feeds coming into the system), must be a simple, low effort activity.

The resulting system, which we call Ningai (a small Australian mouse-like marsupial), is based on a cluster of hefty PCs running Linux, connected by a high speed network. In the following sections, we briefly describe the business problem and its system requirements. We then describe the Ningai architecture and how it meets its goals, and the current implementation, both what it is, and how well it has met our expectations. Finally, we'll do some comparisons with Gecko, and discuss future plans.

2. The Problem

2.1. The Business Problem

The flow of records through AT&T's billing systems, from recording to settlement, is fairly complicated. For most business calls, the records flow through seven major billers comprising 15-20 major systems and are processed by a few hundred different processing steps. Complexity arises not only from that inherent in the work (e.g., how to handle records sent to the wrong biller), but also from the tendency to implement new features by tacking on new systems, rather than reworking and integrating the new features into the existing systems. This flow undergoes considerable churn both at the process level, and at the architectural level. The key business problem is: in the face of this complexity and change, how do we know that every call is billed exactly once? (Gecko answered a similar question for residential calls, but business is harder to do for two major reasons: 3 times greater volumes, and probably 6 times as much architectural complexity. Furthermore, each of the major billers was developed and are run by distinct, independent, and competing organisations.)

Ningai attacks this question the same way Gecko did; it tracks the progress of (or records corresponding to) each call throughout the billing process by tapping the dataflows between systems and within systems. This is a data-intensive method; for Gecko, it involved a daily average of 3100 files totaling 250GB per day. It is a measure of how far things have advanced in 5 years in that for Gecko, this was a novel idea, whereas today, it seems only a little excessive. The main difference between Gecko and Ningai is the change in platform from a large multiprocessor system to a loosely coupled cluster.

2.2. The Technical Problem

The problem is fourfold: we need to convert the various dataflow taps into canonical fixed-length *tags* (parsing), we need to match tags for a call together into *tagsets* and maintain them in a database (update cycle), we need to generate various reports from the datastore (report generation), and we need a scheme for backups.

Mainly because tagsets have variable length and determining their reporting status requires quite complicated logic, we are unable to use conventional databases and in fact use simple sorted flat files. To keep these files, or database partitions, manageable, we split the database into about 5000 pieces, based on a hash of the originating telephone number for each call. There are several more requirements, outlined in [Hum00], but they are unimportant here, other than we need to do the daily update cycle as fast as we can.

3. The Current Architecture

At a basic level, the fundamental task of the cluster is to support data storage and management, and to run jobs on the data, and to do so in a scalable, available way. We made some initial structural decisions:

- the cluster is a loosely coupled federation of nodes, and even though there was a leader node for coordination reasons, the control paradigm is that nodes announce resources or ask for work from the leader, and the leader never unilaterally imposes work.
- all data is stored locally at each node; there is no networked storage (like SAN) and certainly no network file system. We distrust both in terms of performance modelling and reliability.
- all activities, such as job executions, and data storage, have leases. All cluster activities and facilities must support nodes going down and being added with fairly minimal impact.
- we assume the presence of a very fast and scalable networking fabric.

The cluster infrastructure has two fundamental concerns, data storage and job execution, and support for high availability; these three issues are described below.

3.1. Data Storage

There are two kinds of cluster data. The first is various configuration files and the like; these are distributed from the primary source machine via mechanisms equivalent to the *rsync* program[Tri96]. The second are all the various application data files, such as feed files, tags to be added to the database, and the database partitions themselves.

These application data files are managed by the *replication manager*, or *repmgr*. This is a user level, file based, replication service that distributes copies amongst nodes, somewhat similar to, although substantially simpler than, other such systems such as Ficus[Pop90] and Magda[Wen01]. Although current fashion favours schemes replicating at the system call (read, write) level, we felt a user level scheme was easier to manage, and easier to diagnose when things go wrong.

Repmgr handles a single database, namely that of registered files. Files have a simple data view; they have a simple, nonhierarchical, name, MD5 checksum, length, replication count, and a callback mechanism (called when the file is correctly replicated). Files are referred to by (name,md5) tuples, and these tuples are unique across the cluster.

Each node maintains its own database of the files, or *instances*; the database relates pathnames and MD5 checksums. Periodically, each node sends *repmgr* its list of instances (of files), together with a lease time, and measures of how much space is available.

Repmgr's work is fairly simple. It takes the list of registered files, and the set of instances from all its nodes, and does the appropriate actions. These are recorded internally as attempts with leases (the copies have leases, the deletes do not); any copy that times out is retried on another node if possible. *Repmgr* logs all its registrations and periodically checkpoints (every 5 minutes); restarting takes 30-120s to rummage through the log, and another 15-20s to start running.

Repmgr also takes hints. These are not allowed to interfere with correctness, and can be used to move files around safely. For example, if a file is replicated on *ning03* and *ning12*, and we give the hint to not store a copy on *ning03*, then *repmgr* will make a copy on another (the least full) node, and only after that copy is made, will it remove the copy on *ning03*.

We are paranoid; every file copied to a node has its MD5 checksum calculated. If a file gets copied and ends up with a different checksum (for whatever reason), *repmgr* will observe a new, unrelated, instance appearing and simply try again. Cleanup scripts are run daily to clean up apparent detritus.

3.2. Job Execution

Somewhat to our surprise, we have a three level hierarchy of programs to manage and execute jobs across the cluster. It is, however, quite a robust arrangement.

The bottom level is a per system batch scheduler (*woomera*) inherited from Gecko. This is a simple and flexible engine which supports a number of constraints, such as maximum load and number of simultaneous processes, in addition to arbitrary constraints, called *resources*, which are akin to counting semaphores. Although any node can submit requests to *woomera*, all subsequent action takes place locally and independently of the cluster.

The next level in the hierarchy is *seneschal*, which implements cluster-wide job executions, handling node failures and doing some low level scheduling optimisations. *Seneschal* takes job descriptions with three important characteristics: how to invoke the program, the input files/resources needed, and the output files generated. On the basis of resources declared by nodes, *seneschal* allocates jobs to these nodes and schedules the jobs via *woomera* on those nodes. The model is that jobs are posted (on *seneschal*), and nodes bid for these jobs.

Node failure is handled by each execution having an assigned attempt number and a lease. The output filenames for that execution have the attempt number embedded in them. Upon successful execution, the output files are renamed and the job is regarded as having succeeded. If the lease expires, or the job otherwise fails, then *seneschal* reassigns it (to another node if possible), incrementing the attempt number. If multiple attempts succeed, one is chosen as the successful execution and the other(s) are treated as though they failed. This scheme relies on the fact all jobs execute locally and may only affect local, and not cluster-wide, state.

```

%BUNDLE = /ningai/poot
%DATE = 20010728

step2a: [%p in <partitions(abs)>] "generate add file"
    <- %node_add
    -> %add = %p-342.add
    cmd ng_add_gen -i %input %p

step2b: [%p] "partition update"
    <- %add
    <- %oldpart = %p-341
    -> %newpart = %p-342
    -> %delete = %p-342.del
    -> %report = %p-342.rpt
    cmd ng_pu -d %DATE -e 342 %oldpart %add %newpart %delete %report

step3: [%p] "report for deleted tags"
    <- %delete
    -> %delreport = %p-342.delrpt
    cmd ng_report_step1 -e 342 -d %DATE %delete %delreport

```

Figure 1: A *nawab* fragment.

Seneschal also supports a number of convenience features such as specifying a node to run a specific tasks on, and also supports *woomera*-style resources to help manage job streams. *Seneschal* runs as a single copy daemon, with no checkpointing; *nawab* feeds *seneschal* all its input and *nawab* does its own checkpoints.

The final and top level in our hierarchy is *nawab*. This is both a language, and a daemon supporting management of jobs specified by *nawab* programs. *Nawab* is a small domain specific language designed to facilitate handling large interrelated job streams; figure 1 shows a fragment of the program to perform an update cycle of the Ningai database. A full description of the syntax is beyond this paper but the highlights are

- iterators are enclosed in `[]`; they are typically are partitions or sets of nodes. The partition set here is specific to the *abs* application.
- inputs and outputs are denoted by `<-` and `->` respectively; in addition, they can be given symbolic names usable in command strings.
- definitions of variables and iterators stay in effect until redefined

Note that *seneschal* handles the issues of sequencing and ensuring inputs are made prior to initiating jobs.

Nawab is not just a front end or compiler for *seneschal*; it also supports managing (deleting, pausing, monitoring) jobs in terms of the actual

nawab specification. *Nawab* runs as a single copy daemon and does its own checkpointing.

3.3. High Availability

We have a simple view of, and method for, high availability. All programs fall into two classes:

- ephemeral executions should either fail or succeed. In general, temporary resource issues should be handled by failing immediately, and letting the surrounding retry software do its job.
- long running executions, such as daemons and programs like *nawab* and *repmgr*, should checkpoint periodically and log every change pertinent to restarting in the master log file. Both the checkpoints, and as we described below, the log messages, are replicated to all nodes, not only for increased safety, but also so facilitate quicker restart when we change leader. Furthermore, these programs should be embedded within a startup script that execute these programs in a loop, so that there is automatic restart on a failure.

In order to support this model, we take a lot of care over logging. Many programs log into their own logfiles, but all programs log into the master logfile. Logging to the master file is done via both of two methods:

- the log message is written directly to the master log on the current node
- the log message is broadcast (via UDP) to all nodes in the cluster. The logging routine

returns only after acknowledgements have been received from a sufficient number of nodes. If insufficient acknowledgements have arrived before a timeout, the message is written to a local nak logfile. The logging daemon writes messages to both the master logfile and to a secondary logfile.

Periodically, typically every 5 minutes, a process gathers up the secondary logfiles (and nak logfiles if any) and creates a node specific log fragment which is registered with *repmgr*. A few times a day, a second process gathers up log fragments and absorbs them into larger log fragments (one per week for the whole cluster).

Although we have tried really hard to tolerate individual failures of various programs and daemons, sometimes the system is in real trouble and requires human intervention. We do this through a two part monitoring system. The first part periodically checks for processes it knows should be running and if they are not, it drops a critical log message in the master log. Applications may also put such messages in the log, although mostly we would expect just warn and error messages. The second part is a single process *crit_alert* which looks for critical log messages and exits when it sees one. This process is registered with whatever monitoring software our operations support folks use (for example, in Gecko, they used BMC Patrol), and the absence of this process will generate alarms and cause people to get paged. We have auxiliary commands to extract critical log messages to quickly determine what the relevant log entries were. Critical log messages also include a code which identifies the scenario, likely cause, and normal fixup procedures.

4. The Current Implementation

4.1. Hardware

We use fairly regular hardware, except we went for the highest density of disk in our PCs. The configuration is either 1 or 2 Pentium III CPUs per 2U high box. Each system has 512MB memory per CPU, integral 100BaseT Ethernet, a 3WARE PCI disk controller, and an Emulex (previously GigaNet) CLAN network card. Apart from a floppy drive and a sprinkling of CD-ROM drives, the rest of the space in the enclosures is filled with 82GB IDE drives (currently being upgraded to 160GB drives). In order to support experimentation on the effects of CPU speed and multiprocessoriness, we have 16 nodes; 4 of each

combination of 1 or 2 CPUs and 933 or 866MHz processors. We shipped the CLAN network cards to the supplier (Net Express)[†] and they shipped us the systems ready to go with Linux and drivers installed. The dual CPU motherboards also have dual SCSI ports that we use for attaching tape drives.

The other hardware is a CLAN 5300 switch (30 ports) and a QUALSTAR 36180 tape library. The tape library is used just for rolling backups of the database. We backup the source and system images over the network to one of our local servers.

We use WTI power controllers; they were about the only controller available that could supply enough power. (Even with delayed turn on, we can only run 4 systems per 20A circuit.)

At the current time we are using RedHat 6.2 and in the middle of transitioning to RedHat 7.2. Although our intent was to support a heterogeneous set of operating systems, originally, there was only CLAN support for Linux. (Of course, now the best driver is that for FreeBSD!)

We make use of the AST distribution[Fow00] which apart from providing us very useful functionality, also insulates us even more from the underlying system. (The AST distribution includes both a sophisticated set of libraries and POSIX compliant versions of the main user level commands. Perhaps most importantly, the four main authors of AST have their offices within 30 yards of ours, and we have their home phone numbers!) We anticipate very few changes should we move to another OS, such as FreeBSD.

All the knowledge and use of the fast networking is localised to one command *ccp* (cluster cp).

4.2. Software

The rest of the implementation, including the programs *seneschal*, *nawab*, and *repmgr*, work as described above. The rest of the application specific software is largely ported from Gecko and is uninteresting for the purposes of this paper.

[†]Net Express (<http://www.tdl.com/~netex/>) offered to install the cards for us and we're glad we did, because the cards were larger than we expected and Net Express had to change the boxes to make things fit.

5. The Results So Far

This is a work in progress; we are just ramping up into production and we have only just started finetuning things. Nevertheless, we've learned quite a bit already.

5.1. Hardware

We are generally quite happy with the hardware. It is amazing to have one rack contain 16 nodes totaling 24 CPUs and 96 82GB drives, 2 power controllers, an Ethernet switch, and a CLAN switch. While the density is great, we are a little concerned about heat (despite assurances to the contrary); we have seen a few flaky components. We are often asked why we have so much disk per node, and shouldn't we be using RAID. In retrospect, this was a great decision. For our sort of applications, a CPU per 250-500GB of disk is a good ratio; significantly different ratios would yield systems either CPU or I/O constrained. And having this in a single box simplifies scalability calculations; if you buy enough boxes to get the disk you need, you automatically get enough CPU.

Processor (year)	kr/s	/SPARC
250MHz UltraSPARC (97)	56.7	1.00
197MHz R10000 (98)	72.8	1.28
866MHz Pentium III (01)	97.7	1.72
933MHz Pentium III (01)	102	1.80
250MHz R10000 (02)	104	1.81

Figure 2: Relative performance for CPU intensive jobs (parsing) in kilorecords per second.

While we are still investigating ways to increase effective file I/O performance, we are delighted with the CPU performance. Figure 2 shows the performance relative to the two other systems we have measured it on. Please keep in mind that the Sun and SGI systems are 3-4 years old; undoubtedly, their current offerings are faster. For our cluster of 16 nodes and 24 CPUs, the overall throughput was about 1.33 that of the 32 CPU Sun E10000 that Gecko ran on. On the other hand, the cost of the Sun and disk was about 20 times the cost of the Ningau cluster. All hail Moore's Law and the economics of commodity hardware!

Another goal of our setup was to assess the importance of minor differences in clock speed in a production environment (as opposed to one of the standard benchmarks). Figure 3 shows the effective speed of the system for each of the four

cases (1 or 2 CPUs, 866 or 933MHz); the speeds are normalised per CPU. It is rewarding to see we get better speeds from faster CPUs, although the increase is not as great as the increase in clock rate. We expect the modest difference in performance between the single and double CPU case is due to the different motherboard and memory controllers.

CPU	1 CPU	2 CPU
866MHz Pentium III	97.7	98.4
933MHz Pentium III	102	105

Figure 3: How CPU speed and configuration affect throughput in kilorecords per second.

One area where we still need refinement is how to arrange the disk space. The 3WARE controller is a pseudo-RAID controller, that is, it does just striping and mirroring. Originally, we configured it to present each disk as a single LUN, as in our experience, this leads to the most efficient use of the most precious resource (disk heads). However, two problems are forcing us to change this. The first is bizarrely large tap data files; the largest one we have seen so far is 140GB. While we deal with these files in a compressed form (around 10-20GB), it would be nice to have a filesystem where we can store one of these uncompressed. The second is lack of disk speed, which you would normally attack by striping. However, the effects of striping are obscured by the mandatory Linux buffer cache. We're still working this issue, but frankly, this seems a significant limitation for Linux.

5.2. Networking

As we described above, we use the 100BaseT Ethernet for control messages and UDP broadcasts of log messages. This network is comfortably loaded; the average utilisation is under 0.1%, and we've never lost a UDP packet in several months of operation.

Essentially all our data traffic moves over the CLAN network. With the version 1 drivers (on a 2.2.17 kernel), the CLAN behaved itself admirably, being both fast and reliable. Without tuning, we measured 80-100MB/s node-to-node (memory to memory), and often 25-35MB/s. Figure 4 shows the distribution of observed speeds of about 182,000 transfers. Basically, this is a function of how much buffer cache is available. If there is little to none at both ends, you get 4-12MB/s; if there is little to none at just one end, you get 20-30MB/s, and if there is plenty of

buffer cache at both ends, and the source file is in the buffer cache, you get 50-60MB/s. The fabric is circuit-switched and we never observed traffic to any node affecting the transfer speed among other nodes.

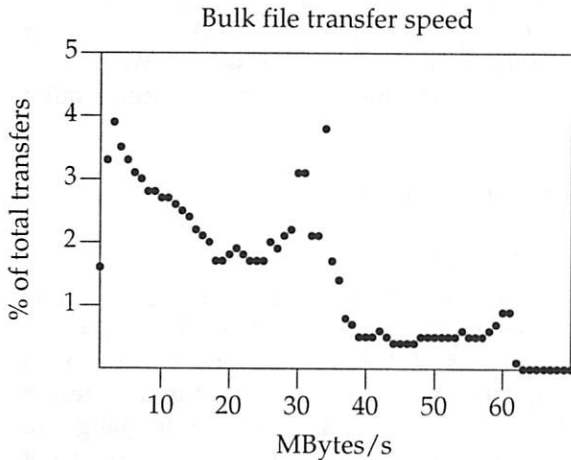


Figure 4: Effective CLAN performance for files larger than 50MB.

Unhappily, we were forced to migrate to the version 2 drivers (we needed to go to a 2.4 kernel for large files and the version 1 driver won't work on a 2.4 kernel). Regrettably, the version 1 and 2 drivers cannot coexist on the same switch (the drivers download microcode into ASICs on the controller card which talk directly to the switch). The version 2 drivers are not nearly as reliable; in particular, rebooting a system sometimes breaks the CLAN network on other systems (requiring them to be rebooted). And Emulex has not been very good about support, although the driver source has been released as "free software" and this is helping a fairly active user community improve on things. As far as we can tell, if you have this hardware, the most reliable (version 2) drivers are now those for FreeBSD[Mag02].

It wasn't an option for us, but if we making the decision today, we would probably go with 1000BaseT Ethernet, although that would introduce some scaling issues (not every node could be on the same Ethernet). There are other networks based on the same ideas as CLAN, such as InfiniBand†, but we know little about them.

5.3. System Software

As described above, we were forced to use Linux because of the CLAN network driver issues. We didn't view this as a problem, rather, we saw it as an opportunity to evaluate whether Linux was ready for prime time. We started with RedHat 6.2 and upgraded the kernel to 2.2.19. We also used ReiserFS for all the filesystems where we store data. It is fair to say we have been surprised at how problematic Linux has been to date (for the record, the authors have predominately used Solaris and Irix); the problems we've seen include:

- we use a scheme for distributing software that used dump/restore. A distribution of the root filesystem involved rebooting to a backup filesystem, *mkfs*ing the old root disk, restoring the distribution onto that disk and then rebooting again. Doing this by hand never failed, but doing it automatically failed about 5% of the time (for no reason that we could find).

- for a while, we couldn't generate new kernels. We needed a special Ethernet driver from Intel that only existed as a module. Apparently over time, our kernel had been growing because we came to a point where it was too big to build as a modularised kernel. (Changing to 2.2.19 fixed this.)

- part of the Linux community's doctrine is the mantra of if you don't like a bug you have, try another release of something (most often, the kernel) and see if it goes away *and* the bugs in this new release don't hurt you. This scattershot, hunt and peck method of trying new kernels until you're happy seemed, and still does seem, odd and unsatisfactory to us.

- we initially used *gdbm* for the per-node database of files. We gave up quickly; despite promises to the contrary, whatever locking it uses was ineffective for our setup. The mean time to a corrupted database was about 5 hours. File locking has been much more reliable.

- we encountered a bizarre problem with a server that did not keep up with a torrent of connections. On any other Unix-like system we've seen, this would fail in a polite way, that is, eventually, the clients would start getting connection refused or somesuch. Instead, no connections failed and the server would get the file descriptor from the *accept* system call as normal. The bad thing was that when loaded, reading the (approximately 100 bytes of) data from the client would take anywhere from 30-400 seconds. As always, the experts recommend trying another

†see <http://www.infinibandta.org> for more details.

kernel, but instead we went to multi-threading the server. (Of course, we didn't thread the server itself as our kernel didn't support core dumps of threaded programs; instead, we built a multithreaded front end that simply dumped all its requests down a pipe to the real server.)

- unlike most other Unix variants, Linux does not support direct I/O. That is, all I/O to files goes through the buffer cache. For applications like ours, file I/O falls into two camps; one needs to be buffered (logfiles, executable images, config files), the other (database partition files) should not. For the sake of performance, and not needlessly churning the buffer cache, we'd like to bypass the buffer cache for the second camp. And it is pointless to say we should have more main memory; more memory always helps, but we'll always have more disk than memory, and in our application, we tend to read and write all the data with very low cache hit rates.

- another consequence of the mandatory buffer cache is enormous variability in the execution times of commands like *df* during heavy I/O loads. In these cases, where a node has a dozen or more files coming in over the CLAN, we have seen *df* take over 30 minutes (only 6 filesystems, no NFS). This is not a huge problem, but some of our scripts cannot tolerate this variability and so we have to maintain a cached copy of the *df* output.

- when the system is heavily loaded, we have noticed that jobs of the form

```
gzip -d < file.gz | a.out > outputfile
```

have about a 2-5% chance of *gzip* complaining about a bad CRC (from a file that is known to be good). We don't care particularly, as our software essentially treats this as a soft error and retries the job again, but we suspect most people would correctly view this bug as pretty unsatisfactory.

- it turns out that ReiserFS has not been a good choice for us. (We chose it on the advice of local experts.) It is unsuited for our file size distribution (relatively few, big, files) and we get a panic in the ReiserFS code every couple of weeks. (This might be resolved by the newly usable *reiserfsck* software.) Having a logging filesystem is really great, though, for quick reboots. We will switch to the *ext3* filesystem when we finish our migration to the 2.4 kernel.

- we were severely constrained by the default (and undocumented) limits on how fast *inetd* can initiate processes. But at least we could look at the code and figure out that there was

this limit and how to change it (which we did from 40 processes per min to 5000).

- we use Java for our monitoring software. The standard distributions ship with *kaffe*, yet if you ask around, *kaffe* is deprecated and has been for more than a year. It would seem more efficacious to ship something that works, rather than let the users find out for themselves. We use the IBM Java environment for Linux; it seems quite solid.

5.4. Cluster Software

5.4.1. Repmgr

The current replication manager is our throwaway prototype. It is about 3000 lines of C and 500 lines of Kornshell scripts. It works well enough that we are taking our time to design and implement the "real" one, despite going into production. As we had learned in Gecko, one of the more useful features is an explain mode, where *repmgr* explains exactly why it is doing, or not doing, every external action (copies and deletes).

While the details of the new version are peripheral to this paper, some are motivated by a particularly good idea we borrowed from the Venti storage server[Qui02]. The core idea behind Venti is that the name, or address, of a block of data is a function of its contents alone. Venti uses a cryptographic checksum (SHA1, although MD5 is equally useful) for its function. What has this to do with the replication manager? Currently, instances are stored on nodes as files whose basename is the same as the registered file. The per node database correlates each instance and its checksum. This is not too bad a scheme, but there are a couple of drawbacks: you have to be careful not to overwrite an instance of the same name, and it can take a long time to recreate the database by recalculating the checksums for all the instances on a node. The Venti idea suggests a better solution: name each instance by its MD5 checksum! There is now no problem with collisions (if the names match, so do their contents!), and rebuilding the database is now trivial.

It is worth asking the question how well does the registry of files correspond to reality? In a deep sense, the registry *is* reality — it is simply the list of registered files. The correct question is how well does *repmgr*'s view of file instances correspond to reality? Like all databases of physical inventory, despite best

intentions, the per node databases of the files on that node seem to start decaying as soon as you create the database! You might call this hubris, but instead, we run a process that corrects and makes the database and underlying filesystems consistent. This runs once a day and as a result, nearly all nodes have zero inconsistencies.

5.4.2. Nawab and Seneschal

The source for *nawab* and *seneschal* is about 10000 lines of C. The charming thing about servers bidding for work is that they are inherently good at load balancing; figure 5 shows a batch run of 700 jobs distributed over 9 nodes. As you can see, we keep all 9 nodes busy for the first 48.5 minutes and then the number of nodes drops off as we run out of jobs. Part of the reason for the sharp drop off is that *seneschal* orders job assignments by size (largest first) so as to get the best amount of parallel work.

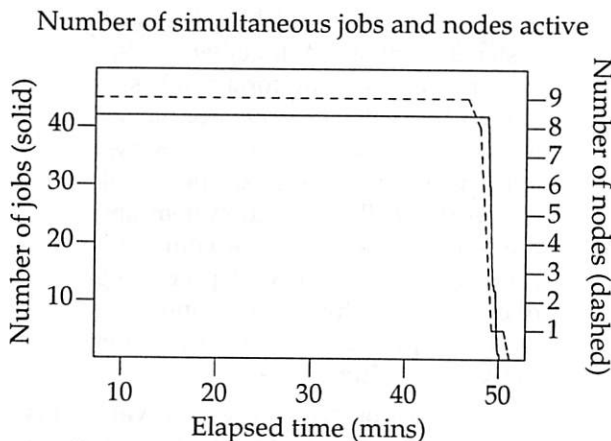


Figure 5: Effectiveness of *seneschal* scheduling.

5.4.3. Elections

Every 5 seconds, the nodes in the cluster elect a leader node using a nifty algorithm by Christof Fetzer, a variant of the algorithm in [Fet99]. (The algorithm involves each system keeping track of each system voting, their priority, and a lease on the current election. When the lease expires, we assume the leader has crashed and the system with the next highest priority will attempt to become leader. The whole election is resolved in just one round of 'voting'.) This may seem fairly frequent, but we only change something if the election yields a new leader. We added two rules to the original election scheme: re-elect the current leader unless it doesn't vote, and if changing leaders, we have specified a preferred winner (again, if that

system voted). We normally specify a 2 processor system to be our normal leader (because of daemons like *seneschal* and *repmgr*).

5.4.4. Logging

Everybody knows it, but most do not do it: log everything! We expect to be generating over 120MB of master log file per day, and that is independent of daemon specific logs. And for every time we have thought we log too much, there have been multiple disasters and weirdnesses which could only be resolved by picking through the logs. Our decision to make the master log file replicated throughout the cluster has really facilitated diagnosis of multinode problems.

5.5. Application Code

Nearly all the application code was ported from Gecko without incident. The main surprise has been in the diversity of data feeds; one system sends us about 1000 small files per day, while another sends us one 140GB file per month! And it is just plain awkward to deal with a 140GB file.

5.6. Other Stuff

5.6.1. Zookeeper

Monitoring and managing a cluster of nodes is not substantially more difficult than managing an SMP environment, but can be much more tedious. While we do not view the cluster as a single (virtual) system, we do require that we can manage it through a single user interface. This user interface, which we call *zookeeper*, presents customised views representing various aspects of the cluster. There are four basic parts of the infrastructure supporting *zookeeper*:

- various scripts that gather various system statistics and details, which are then processed into *view data*. View data is distributed by HTTP.
- Java classes, or *views*, which display view data. Views may represent hardware/system information (disk usage, load averages) or daemon specific information (*seneschal* queue and load distribution information).
- a broadcast mechanism which supplies low latency incremental updates to view data (via UDP/IP packets).
- a method for hierarchically constructing a new view from an arrangement of other views.

Zookeeper's user interface provides a simplistic, but useful, ability to remotely control cluster components by allowing for command buttons within a view. When 'pressed,' a command button executes a preprogrammed command which can allow the user to do such things as start/stop/pause critical processes, pause or resume operations on a node, and even power cycle a node without the user needing to physically log into the machine.

The graphical user interface portion of *zookeeper* was implemented as a Java application, rather than a pure Xwindows application. While less efficient and rather more quirky, this empowers the unwashed masses (who do not run X) to see what is going on. The user interface was not written as an applet as it has the need to retrieve and communicate with hosts other than the host that would supply the applet; something not allowed under the definition of an applet.

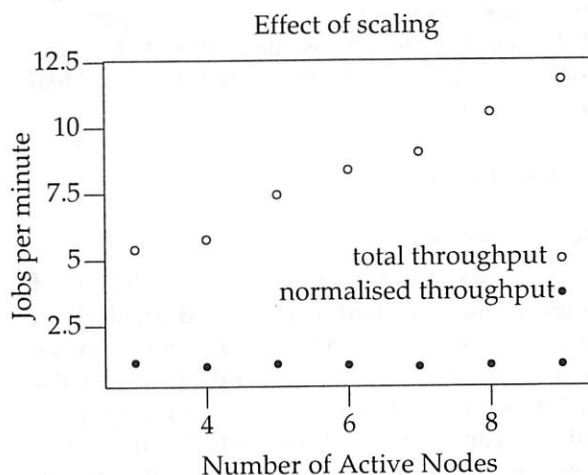


Figure 6: Experiments in scaling.

5.6.2. Scalability

Ningau is inherently scalable because of its loosely coupled design, especially as it's unit of design, the node, is fairly well balanced in terms of CPU and I/O. Figure 6 shows how the performance scales with the number of nodes. It shows both the total throughput, as well as the throughput divided by the sum of the throughputs of the nodes involved. This normalised throughput is relatively constant. The current hardware and software architecture could scale to about 100 nodes comfortably. Above that, we have some issues:

- networking. The CLAN hardware can handle fairly large numbers of nodes but

eventually, any single network will not scale. This mostly affects the replication manager, and in our new design, we are thinking of supporting subgroups of nodes with peer-to-peer interfaces with other subgroups. By having different subgroups using different network fabrics, the network can be increased to a large size, at least up to the thousands of nodes.

- replication manager. *Repmgr* gets sent file lists from all its nodes every several minutes. This does *not* scale well! On the other hand, the above subgroup functionality would also mitigate this problem, as the total number of nodes would increase as the square of the *repmgr* limit.

6. Epilog

One of the unexpected results of the Gecko work was a relationship between high performance architectures for SMP machines and networked clusters:

"designing for a scalable cluster of systems networked together is isomorphic to designing for a single system with a scalable number of filesystems. Just as with a cluster of systems, where you try to do nearly all the work locally on each system and minimise the inter-system communications, you arrange that processing of data on each filesystem is independent of processing on any other filesystem." [Hum00]

The foremost goal on Ningau was to verify this relationship and to evaluate whether or not a loosely coupled cluster of Linux systems was in fact competitive with large, industrial strength, high end SMP servers.

Our evaluation is not over; we have still to go through the grind of doing production runs for several months before we know for sure. But so far, the answer is yes. Despite our litany of problems with Linux and the GNU software, it is a smaller and less serious list than what we faced with Sun and Solaris in Gecko. And in almost all cases, the workarounds were fairly easy, even if tedious. They have also reinforced our professional grade paranoia; we believe in checksumming, and checksumming often. (Recently, we discovered that the *zlib* compression library does not detect and pass through *all* I/O errors, and as a result, a file system running out of space was not detected and some poor operator had to resend us several thousand files.)

Even if we judge the software reliability issues as even, the tremendous cost, availability, and scalability advantages of the cluster are just too great. Eventually, clusters will rule the business world, particularly the medium to high end.

Glossary

- bid* A request for an amount of work. Sent by a node to seneschal.
- biller* One or more programmes responsible for calculating a customer's bill based on information recorded by the switch for each phone call.
- cycle* The process which takes all of the tags received during a period of time (usually 24 hours) and updates the database with them, deleting old tagsets, and generating reports.
- feed* A series of data files received from the same source. (See stream)
- hint* A suggestion given to an application such as *repmgr* regarding how it should manage something within its domain. Hints are not mandatory and may be ignored if complying with the hint would affect the correctness of the domain.
- lease* A period of validity for things such as hints, and jobs. The application managing an item with a lease must assume that the item (hint, job, etc) is valid until the item is specifically changed, or until the lease expires. A lease expires when its endpoint is no longer in the future.
- parsing* The process by which records within a data file are converted to tags.
- stream* A group of feeds which are related and can be processed in the same manner.
- tap* A point in the billing flow where data is syphoned off and sent to Ningau.
- tag* The set of information that is extracted from each data record. A tag represents a single phone call within the billing flow at a single instance in time.
- tagset* A group of tags which represent the same phone call. The tagset describes the history of a phone call as it was processed within the billing environment.

Acknowledgements

This work was a team effort; we have had vital help from Angus MacLellan and assistance from Christof Fetzer, Mark Plotnick, and Debi Balog.

The comments of the reviewers and our shepherd improved this paper; the remaining errors are those of the authors.

This is an experience paper, and as such, contains various statements about certain products and their behaviour. Such products evolve over time, and any specific observation we made may well be invalid by the time you read this paper. Caveat emptor.

Contact Information

For any more information about this paper, or the software described, please contact Andrew Hume. We expect to be able to release some of the software tools described here, but the details will vary over time.

References

- [Bec95]. Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer, "BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION," *International Conference on Parallel Processing*, pp.20-25, IEEE, <http://www.beowulf.org/papers/papers.html> (1995).
- [Fet99]. C. Fetzer and F. Cristian, "A Highly Available Local Leader Service," *IEEE Transactions on Software Engineering* 25(6), pp. 603-618, <http://www.research.att.com/~christof/HALL> (Sep 1999).
- [Fow00]. Glenn S. Fowler, David G. Korn, Stephen C. North, and Kiem-Phong Vo, "The AT&T AST OpenSource Software Collection," pp. 187-195 in *USENIX Conference Proceedings*, USENIX, San Francisco (Summer 2000).
- [Hum00]. Andrew Hume, Scott Daniels, and Angus MacLellan, "Gecko: Tracking A Very Large Billing System," pp. 93-105 in *USENIX Conference Proceedings*, USENIX, San Francisco (Summer 2000).
- [Mag02]. Kostas Magoutis, "Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD,"

BSDCon02, San Francisco, pp. 65-76 (Feb 2002).

[Pop90]. Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann, "Replication in Ficus Distributed File Systems," *Workshop on Management of Replicated Data*, pp. 20-25, IEEE (Nov 1990).

[Qui02]. Sean Quinlan and Sean Dorward, "Venti: a new approach to archival storage," *Conference on File System and Storage Technologies*, Monterey, pp. 89-101, USENIX (Feb 2002).

[Tri96]. Andrew Tridgell and Paul Mackerras, "The rsync algorithm," ANU Computer Science Technical Reports TR-CS-96-05, Australian National University (1996).
<http://rsync.samba.org/>

[Wen01]. Torre Wenaus, BNL (2001).
<http://atlassw1.phy.bnl.gov/magda/info>

CPCMS: A Configuration Management System Based on Cryptographic Names

Jonathan S. Shapiro John Vanderburgh
shap@cs.jhu.edu vandy@srl.cs.jhu.edu
Systems Research Laboratory
Johns Hopkins University

Abstract

CPCMS, the Cryptographically Protected Configuration Management System is a new configuration management system that provides scalability, disconnected commits, and fine-grain access controls. It addresses the novel problems raised by modern open-source development practices, in which projects routinely span traditional organizational boundaries and can involve thousands of participants. CPCMS provides for simultaneous public and private lines of development, with post hoc “publication” of private branches.

This paper describes the repository architecture of CPCMS, and in particular its use (and abuse) of cryptographic naming mechanisms to achieve collision-free disconnected operation.

1 Introduction

CPCMS, the Cryptographically Protected Configuration Management System, is a new configuration management system that provides scalable, distributed, disconnected, access-controlled configuration management across multiple administrative domains. All of these features are enabled by the pervasive and consistent exploitation of cryptographic names and authentication.

Software configuration management (CM) systems provide multiple developers with a consistent, shared view of a project that is changing and evolving. When development occurs in geographically dispersed locations and is split across multiple organizations, issues of access control, integrity management, update distribution, and high-volume change integration become significant concerns in choosing a CM system. This is especially true in open source projects. In addition to the “core” team, which may consist of developers from several companies, an open source project may have small patches submitted by thousands of users that need to be integrated. As open source projects become more successful, the demands placed on the supporting CM tools grow with them.

In the most immediate sense, CPCMS was brought about by the expansion of the EROS team and the impending release of the EROS system to outside development groups. EROS [SSF99] is an open source, secure operating system that is currently gaining mainstream interest. From 1991 to 2000, the EROS project relied on the CVS system [Ber90] for its configuration management. As the project reached the point where outside developers would be making substantive changes, it became clear that CVS did not meet several of our requirements:

- First-class support for configurations, and in particular for *rename* operations.

- Support for disconnected commit and distributed repositories.
- Scalable, consistent replication. While tools such as *rdist*, *rsync*, and *mirror* provide replication, a user “updating” at the wrong moment can easily end up with an incomplete or inconsistent result. There is no simple way for a user to detect this.
- Per-file access controls. The CVS-ACLs system does not protect RCS logs.
- Support for development groups that span multiple companies or administrative domains. While uncommon in proprietary projects, such groups are nearly universal in open source efforts. CVS provides no authentication mechanism suitable for multi-organizational development.

When only a limited number of developers worked on the EROS code base, these shortcomings were relatively manageable annoyances. As we contemplated the prospect of hundreds or (optimistically) thousands of potential contributors and “lurkers,” these requirements suddenly loomed as serious concerns.

Subversion, a successor to CVS currently under development by Tigris.Org [SVN], addresses many of these issues, but is notably lacking in disconnected commit support or multi-organizational authentication. Other CM systems similarly did not address our requirements.

Ignoring the problem of delta management in the repository, a configuration management system is not conceptually difficult to design and build. The principle architectural challenges in distribution lie in naming and consistency. Cryptographic naming seemed to provide a solution for both problems that we wished to explore. Cryptographic

tographic names uniquely identify the content they represent. They are collision-free, but can be generated without shared access name binding agent. They are also pragmatically impossible to forge. A properly constructed cryptographic name therefore provides an inherent integrity check on the content it names.

Finally, we felt that a CM system based on cryptographic names might provide an interesting basis for future research. In particular, different projects impose different policies on their repository content and often use local customizations (e.g. triggers). We were intrigued by the possibility that a safe programming language might be integrated with the CM client to enable this in a safe, platform-independent way. This is a "future," but the current base will clearly support it.

This paper describes the architecture and the implementation of CPCMS, our early experiences and problems with the first design, and the results obtained by its reformulated successor: OpenCM.

2 Comparison to CVS

CPCMS is ultimately intended to replace CVS, and the command line CPCMS client has a very similar "feel" to the CVS client. Like CVS, the basic CPCMS usage model is to check out some branch of a product, edit it locally, perform integration updates along the way, and ultimately commit the result back into the repository.

CPCMS differs from CVS in several ways:

- To support rename operations, CPCMS maintains a set of bindings from workspace object names (e.g. file names) to objects.
- Versioning is on configurations, not on files. Every commit, whether a single file or a complete replacement of the code base, creates a new configuration object in the repository.
- In a laptop configuration, CPCMS replication can be used to cache the originally checked out version in a laptop-based repository. Status checks and "undo" can be done without connecting to the network.
- CPCMS is workspace-neutral, in the sense that it can support workspaces other than files and directories. For example, the information architecture could readily support an object workspace such as those used by several integrated development environments for Java.
- CPCMS uses cryptographic rather than password-based authentication.

- Commits need not be made to the originating repository. This allows work to be committed – and potentially undone – while disconnected.

The last point is important, as it is the basis for disconnected development. A user can check out a project, make some changes, and can realize after the fact that they are disconnected or for some reason lack the authority to check in those changes. Instead of losing their work, they can instead create a new branch in a local repository, allowing development to proceed. At a later time, after the impediment has been removed, this local branch can be synchronized to the central repository and a merge can be performed.

3 System Architecture

In this section we describe the CPCMS architecture and how this architecture facilitates replication and disconnected development.

3.1 A Client/Server System

CPCMS is a client/server system. The client implements all of the semantics of the configuration management system. The server provides an archival object repository with specialized support for objects that are frequently revised.

Clients use an RPC-based protocol to make requests on the server over an encrypted, mutually authenticated connection provided by the Secure Sockets Layer (SSL). While remote procedure calls do not achieve the best possible bandwidth utilization, they provide a reasonable compromise between client simplicity and effective network utilization. Most of the possible asynchronous exchanges between a CPCMS client and server can be simulated in the remote procedure call interface by providing an interface that supports bulk batch transfer of managed objects. A more asynchronous interface and protocol is under consideration for a future version of CPCMS.

3.2 Naming in the Object Store

Objects in a CPCMS repository are divided into two categories: frozen and mutable.

A **frozen** object is immutable. From the perspective of the repository, each version of a file, each configuration, and each access control list is a frozen object. Frozen objects cannot be modified. Changes to (e.g.) file content in the repository are recorded by introducing new frozen objects corresponding to the new state, and updating some mutable object (see below) to point to the new objects rather than the old ones. This is because a CM repository is

an archival store. While a new version of a file may be introduced, the old version must continue to exist as an independent object for reasons of historical traceability.

A **mutable** object is one whose identity must remain the same across modifying operations. A branch, for example, is a sequence of configurations. As new configurations are added, the branch grows, but the branch itself retains the same identity after the commit as before. When a new configuration has been appended to a branch, we say that the branch has been **revised**.

Frozen objects are named by the cryptographic hash of their content. A cryptographic hash provides a short, unforgeable bit-string that uniquely identifies the original content. Because they are unforgeable, cryptographic hashes are also collision resistant. CPCMS uses the SHA-1 cryptographic hash function for frozen object names [FIP94]. Mutables are named using server-generated **swiss numbers**, which are identifiers generated using a strong random number generator. Mutable names also contain the server's unique identifier. The mutable itself contains the signature verification key of the originating server. The mutable content plus its name are signed by the originating server, and the signature is distributed as part of the mutable.

Both naming mechanisms ensure unforgeability of content. For this reason, we refer to these two types of names collectively as **true names**. Server-generated true names are a pair consisting of the server's unique identity followed by the randomly generated server-relative object ID. The server checks for collision of mutable true names.

CPCMS has no means to recover from a collision of cryptographic hashes. We are aware of no mechanism that (in principle) can provide such recovery without central coordination. The SHA-1 algorithm is a 160-bit hash. In such a hash, the expected number of objects required to generate a hash collision is 2^{80} . As an empirical test, we have obtained copies of several large CVS-based repositories and extracted every version of every file in these repositories. To date, we have not observed a collision except where content was actually identical.

Identical content is the one case in which cryptographic name collision is guaranteed. From the CPCMS perspective, this is not only desirable, it is essential. Name collision on identical objects is the basis for eliminating unnecessary communication. As a practical example, if two otherwise empty files are independently checked in with the same copyright notice, only one frozen object will be stored and both configurations will name the same frozen object. Given that the bits are identical, it does not matter which entity is returned on checkout. Because the CPCMS repository does not interpret the content of its stored objects (other than for garbage collection), the se-

manantics of objects in the repository is completely defined by their content.

Having acknowledged the "Achilles heel" in the CPCMS design, it should also be said that the use of cryptographically generated names has several desirable properties:

- Given two objects with different content, we can generate non-colliding names without appeal to a centralized naming mechanism. This is a fundamental requirement for successful disconnected commits.
- The hash serves as an integrity check that lets us verify the integrity of the object. By recomputing the hash, the client can determine whether the content of the object has been improperly altered.
- Because hashes are universally unique, the hash can be used to avoid network transmission of objects that are already present at the destination.
- Because hashes are sparsely allocated from a very large space, they are pragmatically impossible to guess. This is a crucial underpinning for the CPCMS access control architecture, which is described below.

Using cryptographic object names eliminates the need to deal with case sensitivity, file name length, or path length issues in choosing server-side names. In fact, a file-independent naming strategy divorces the repository implementation from underlying platform dependencies altogether. Repositories can be constructed to use storage strategies ranging from use of an object database to storage within the server file system itself via some transformation on the true name. Further, it is straightforward to construct a "union" repository by which a new repository implementation can be run in parallel with an old one for testing purposes.

3.3 Naming Managed Content

The CPCMS server makes minimal assumptions about the client-side semantics of managed content. Ignoring issues of garbage collection, the CPCMS server provides configuration-based versioning of uninterpreted "blobs" (binary large objects) and records bindings between client-side names (*C-Names*) and internal object names (*true names*). To the server, these c-names are uninterpreted strings. It is entirely up to the client what names to use, and what organizational semantics should be associated with these names and their bound content in the client-side workspace. Examples of possibly valid c-names include:


```
sys/kerninc/Process.h (a file)
org.apache.xalan.xsl.Process (a class)
<Object 0x0040824> (an object pointer)
```

For example, CPCMS does not assume that the objects managed consist of files, nor does it depend in any way on file semantics. While the current command line client is designed to manage workspaces of files, an alternative client might use the repository to manage an object workspace just as easily. As a result, the CPCMS repository (and most of the logic of the file-based client) could be used for workspaces that are not file oriented, such as a Java or Smalltalk class repository. One could also imagine directly binding a CPCMS branch as a namespace that can be exported directly through a web server (we are in the process of building this).

Similarly, the CPCMS server has no responsibility for client side serialization and deserialization. CPCMS delivers the requested blob. Further interpretation is left to the client. Correct client interpretation is facilitated by recording the "type" of the object along with its name binding. For example, the client may record that the object content is ASCII at check-in time in order to know that newline conversion may be needed on checkout. The server knows nothing of client side canonicalization.

The file-oriented CPCMS client stores configurations as a set of (c-name, type, truename) triples. When the CPCMS client processes the triple:

```
(kerninc/process.h, ASCII, truename)
```

it interprets this to mean that the intervening directories must be created if they do not already exist. That is, the client views the existence of directories as an emergent consequence of the need to bind c-names. Directories typically have no first-class existence in the repository. This means that directories are renamed in the workspace as a side-effect of renaming the files that live in them. The client facilitates this by simulating the behavior of the UNIX `mv` command when it is passed arguments that imply that a "directory" is being renamed.

Where it is necessary to ensure that an empty directory appears in the client working tree, a name binding with type DIR can be created. This name binding is handled in the same way as the file name binding shown above, and renaming is handled similarly.

3.4 Naming for Users: Take 1

While cryptographic names solve the repository's object naming problem, they are not terribly useful to human beings. As a user, I want to work on "the main branch of the EROS project", not some strange ascii-encoded random number.

To facilitate human association, the original CPCMS ar-

chitecture provided a per-server namespace of "pet names." Each project or branch carries with it both a human readable description and a "nickname." When a project or branch is first replicated to a server, the server generates a petname by applying a collision avoiding transformation on the nickname. This provides both a means of human association and a means by which two users on distinct servers can arrive at closely related names for the projects they share in common.

Experience shows that this design does not work. The CPCMS architecture encourages users to make many branches, and the results quickly create a pet name space too crowded to be practically useful. One usability conclusion from this is that users need to be actively involved in name selection for the naming mechanism to be useful. The CPCMS architecture also makes it difficult to supply an equivalent to the CVS "tag" mechanism. In the course of cleaning up this and other problems in the original CPCMS design we arrived at a different and more workable approach, presented in Section 7.

3.5 Replication

CPCMS's scalability is built on replication. Because frozen objects are named by their cryptographic hash, these objects are non-colliding, and mutual replication of such objects is straightforward. For mutable objects differences are resolved by taking the copy with the higher sequence number. This works because of two constraints that are imposed on mutable objects:

- Every mutable object has a single server that is said to "originate" that object. Valid modifications to the object can only be performed by the originating server.
- Each change to a mutable object is signed by the originating server. The signature verification key is itself embedded in the mutable's state. Accuracy of an alleged version can therefore be determined by checking the object signature.

Ironically, the problem in the resulting replication architecture isn't so much deciding what to keep as deciding what to throw away. Client side repositories can rapidly come to hold many object versions that are no longer of interest. At present, a simple garbage collection mechanism that preserves the top N configurations of all un-owned branches is used. Other options are being considered, and this is an open area for future work in CPCMS. One reviewer of this paper suggested "last N days" as a potentially useful metric.

A second reviewer encouraged us to re-examine some of the approaches used in the Elephant file system [SFH⁺99],

which must similarly decide when to forget. The Elephant design works in part because each content version is logically independent. Any object pointers implicated by forgetting a file version are owned by the file system, which is free to update them. This approach does not translate straightforwardly into an archival object system, because objects in the archival store embed pointers to predecessor versions.

CPCMS content is encoded in ASCII XML form, allowing replication across heterogeneous servers without regard to issues of word order or host platform restrictions.

4 The CM Schema

The CPCMS configuration management schema can be divided into two parts: the content schema and the configuration schema. Each content object, such as a file, is implemented as two repository objects (Figure 1):

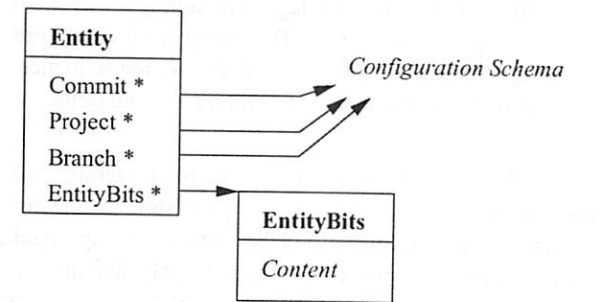


Figure 1: Content schema

Entity An Entity is a single version of a managed object (a file, the text of a function, a document, etc.) Each entity records the identity of the commit description record under which it was created, the names of its containing project, branch, and version, and a list of its immediate predecessors (it can have two – the second due to a merge). It also records the *c-name* of the object, which is the name by which it is referenced in the client environment (e.g. “docs/dcms.xml”).

EntityBits An EntityBits object contains the actual content of an entity version (along with its length). In RCS and SCCS, entity metadata and content are stored in a single file. By separating these in CPCMS, we allow the merge algorithm to trace the evolution of objects without actually fetching the large volume of data associated with the content of each version.

Further, the cryptographic hash by which the EntityBits object is named allows the merge algorithm to trivially check whether the file version that is being merged is identical with the one already in the workspace – which is

the majority of cases. The merge algorithm simply constructs an EntityBits object in memory from the file already present, and checks if the true name of this object is the same as the true name of the EntityBits to be merged.

The schema supporting configurations is more complicated (Figure 2). Every CPCMS commit generates a new CommitInfo object, a new Change object, and an append to an existing Branch object. The CommitInfo record is separated from the Change object so that each Entity can carry the history of its changes and ancestry. The configuration schema must also track the responsible party for each change.

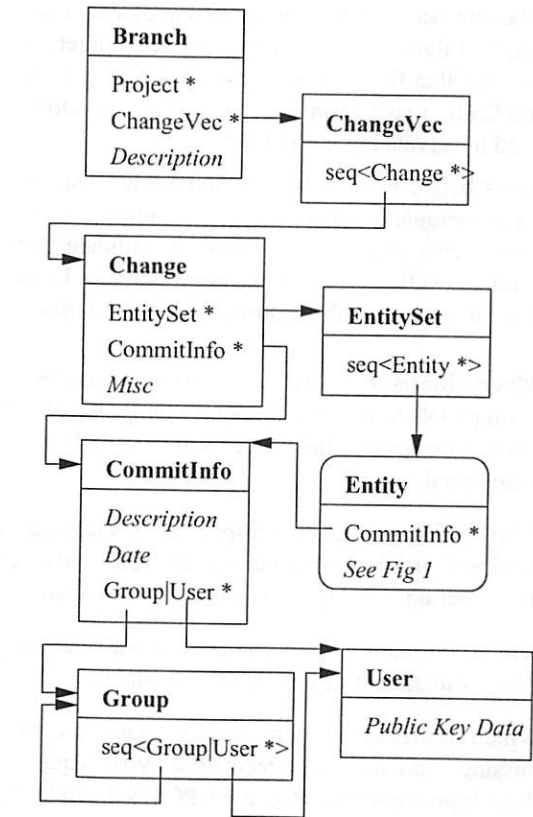


Figure 2: Configuration schema

Project and Branch objects are unusual in that these are mutable objects. Every mutable object carries a “read group” and a “write group” identifying who can read or modify the object. Each Project and Branch object provides a human-readable description of the corresponding project or branch.

The repository storage format for objects includes their type in the first word, allowing tools such as integrity checkers or browsers to be constructed, and supporting occasional garbage collection to reclaim storage from uncompleted transactions.

5 Relationship to Objectives

CPCMS is primarily focused on three problems: scaling, disconnected commits, and access control. Each of these is a source of difficulty in current CM systems. In this section, we discuss how the CPCMS information architecture addresses these issues.

5.1 Scaling

By “scaling,” we mean that we wish to enable hundreds of thousands or millions of users to track updates and revisions to one or more aggregate works. This is clearly beyond the number of users that can be handled by a single server, and doing so is not an objective. Rather, we wish to ensure that the CM schema allows simple incremental replication so that multiple, satellite repositories can be used to serve a large user base.

Connections between machines are sometimes lost, and machines occasionally crash, yielding incomplete transfers. Both clients and servers are able to validate their current content without consulting other agents. Using purely local information, this examination can determine:

- Which objects in the repository, if any, have been incompletely transferred or (equivalently) have been locally corrupted. These objects must somehow be re-acquired.
- Which objects, while uncorrupted, are not referenced by locally visible configurations. These should (eventually, but not too eagerly) be garbage collected.
- Which configurations are complete, in the sense that all constituent entities are locally available.
- Which entities in the client working arena have “gone missing” and should be recovered by re-acquiring them from the server. Because CPCMS (by default) allows file modifications in the working arena without locking, it is usually unable to distinguish between corruption and modification. The rule here is “notify where possible, but don’t make an uncertain fix.”

By combining checks of the true names against the actual content and traversing the reachable objects, each of these concerns can be validated by a client application.

5.2 Disconnected Operation

Under the heading of “disconnected operation” we include two types of usage: revisions committed by disconnected developers and revisions made in private lines of development.

As the storage and processing capacity of laptop machines has increased, their use as development machines has become ubiquitous. The bulk of this paper was first written on a laptop while sitting in front of a fireplace or on various airplanes. The greater part of CPCMS itself was written on various airplanes between New York and Seattle. Neither of these locations is networked.¹ While dial-up connections from hotels are usually possible, they are hardly ubiquitous and are usually quite expensive.

There are two consequences of this type of migratory development:

- Development mistakes are increasingly made in disconnected locations. The ability to “check in” locally in order to have a means of recovery is increasingly important.
- The configurations resulting from these check-ins will later need to be integrated back into the main development repository. This integration should preserve the entire history trail of the change sequence, not just commentary on the final resulting delta.

This problem can be solved by introducing a hierarchy of repositories, as in NUCM [dHHW96], but semi-connected hierarchies require use of a “lock before modify” approach. Locking requires connection, which largely defeats our goal of disconnected commit. Also, we have found in practice that locking is disruptive for mid- to large-sized projects. Header files, in particular, become sources of lock contention. Post-integration, as provided by CVS, is (subjectively) more productive in properly structured projects. In any case, an unrelated aspect of open source development argues against hierarchical repositories: the need to operate under multiple administrative domains.

Open source developers are frequently part time. In the context of their employer they work on one set of projects. In the context of the open source community they work on another. If we imagine that their laptop holds a repository that serves as a local cache, then this repository has two parents: one associated with their employer and one (or more) associated with open source efforts. The hierarchy, if any, is a function of the *project*, not of the *repository*.

The CPCMS solution to this problem is to allow private branches. The laptop user creates a laptop-local repository that mirrors the desired project from the primary server for that project. To develop remotely, a new branch is created in the laptop repository and commits and changes are applied locally. At a later time, when a connection is available, the laptop branch can be re-integrated (via merge) into the main line of development.

¹ The fireplace is now served by wireless, but it wasn’t at the time.

This approach incidentally solves the private line of development problem. The “private branch” can serve equally well as the start of a private, in-house line of development.

5.3 Access Control

The access control design objectives for CPCMS are relatively straightforward. Wherever access control is applied, we wish to distinguish between who can read, who can write, and who can alter the access control rules. The mechanism for access checking should make use of cryptographic techniques so that no local “account” is required on a server to read or write the repository. In part, this is necessary to ensure that mirrors of a repository can enforce access rules without prior knowledge of particular developers. Our current authentication model is provided by SSL using self-signed certificates. The server holds a certificate for each authorized user.

It is assumed that downstream replicate repositories are trusted in a limited sense. Because the replication mechanism does not use a privileged API, replicating servers make copies under the same access constraints as end users. We are not unduly concerned with malicious modification of entities, because the truename system provides a self-checking means of detection. For mutable objects, each change must be signed by the originating repository, whose private key is undisclosed.

CPCMS does not prevent people from making modifications to projects in their own repositories, so long as these modifications cannot be used to corrupt some other line of development. Given that a user can read a project branch, there is clearly no way to prevent them from checking this image into a local repository as though it were a new project of their own. Because of this, there is no strong reason to prevent the creation of private, mutable branches in privately controlled repositories. There are, however, two reasons to encourage this in preference to checking in the code again:

- In some cases, it may become desirable to publish (and possibly to re-integrate) these private lines of development at a later time. By preserving their relationships to the public portions of the project, an orderly merge is greatly simplified.
- If the private effort is in fact a private line of development, the developer can use CPCMS to continue integrating changes from the main line of development into their private version, reducing version drift.

The original design of CPCMS called for access control only at the project and branch level. We quickly concluded that this was unworkable. The main reason for this

is that different portions of a code base may require different degrees of expertise to modify them successfully. Requiring people to work in separate branches to perform restricted modifications places a cumbersome burden on the system integrator.

To avoid this burden, CPCMS includes as part of each branch or project description a text object that specifies for each user or group a set of patterns describing the objects they can modify. By associating this object with the branch, access control propagations propagate as quickly as changes to the branch. For branches and projects, read access changes are rare, typically *increase* access (that is: you don’t delete people from the read list often), and reasonable propagation delays seem acceptable. Write access for branches is centralized by the existence of a single originating server in any case, so there is no propagation delay for write controls. For frozen objects, access control is applied not so much on the object as on the c-name namespace. In effect, the access controls determine which portion of the c-name space a given user is permitted to rebind.

Oddly, CPCMS does *not* implement read access checks on entities. Entities in the CPCMS repository are named by cryptographically secure true names. As a result, they are unguessable. If the user does not already have in hand the object describing the branch, it is impossible for them to successfully guess the true names of entities within that branch.

There is only one case in which this is untrue, which is the case where a legitimate user exposes the entity’s true name to the unauthorized user. We note that in all cases where this might occur the legitimate user could also have passed the entity itself; at modern bandwidths there is no meaningful difference in transmission cost or time. There is a small marginal disclosure in transmitting the entity true name, which is that each entity knows its predecessors. Disclosing the entity (as opposed to its content) therefore discloses the sequence of changes that arrived at the current entity version. Here again, however, the legitimate user could simply transmit the entire sequence.

See the future work section for a proposed solution to this problem.

6 Initial Implementation

Our initial server implementation used a simple file tree structure to store objects:

```
./frozen
./frozen/[hash-1]
./frozen/[hash-2]
./frozen/...
```

```

./petnames/
./projects/
./projects/[rand-3]/
./projects/[rand-3]/branches/
./projects/[rand-3]/branches/[rand-4]
./projects/[rand-3]/branches/[rand-5]
./projects/[rand-3]/petnames/
./projects/[rand-6]/
./projects/...

```

This repository essentially ignores space efficiency issues. Its one merit is that it was very quick to implement. Using a simple structure reduced the entire effort to something that we thought might be tractable to test with bounded effort.

While we were aware that flat files were not the way to go in the long term, this design seemed initially credible. One of the most widely used file version management tools today is RCS [Tic85]. Measurement using the EROS source repository revealed that RCS storage is only 20% more efficient than Lempel-Ziv compression of individual file versions. Since users seem to find the storage cost of RCS acceptable, and the cost of disk space continues to fall, we decided *not* to implement any clever storage techniques in the our first repository implementation.

A second factor motivating our initial decision was XDFS [Mac00], a filesystem-like repository built on XDELTA. XDFS provides most of the repository storage optimizations we want, and already supports cryptographic naming and delta-based compression. Discussions between Shapiro and Josh MacDonald (the author of XDFS) early in the CPCMS design cycle suggested strongly that the two pieces of work would converge at about the right time. As a result of these discussions, it was decided jointly to integrate XDFS into CPCMS in a later release of CPCMS and focus our initial attention in the CPCMS project on schema and usability issues.

As events turned out, our plausibility argument about RCS file sizes was entirely and horribly wrong, for reasons that should have been obvious in hindsight. The issues, and the steps we have taken to solve the problem are described in Section 7.

Fortune sometimes favors the paranoid. Though we chose an initially simple implementation, we were aware that a better implementation would someday be necessary. The repository interface therefore provides a hinting mechanism to support delta storage in the repository. While frozen objects are immutable, the fundamental operation by which a new entity is introduced into the repository is the `revise()` operation, which specifies a (possibly null) hint about the predecessor from which the new entity is allegedly derived. This hint can be used as a basis for delta calculation in storing the new entity. For config-

uration management purposes, entities internally encode their true predecessors where appropriate. The predecessor specified via the `revise()` interface is considered advisory, and the repository implementation is free to ignore the hint in favor of other compression strategies.

7 From CPCMS to OpenCM

CPCMS became fully operational as we were revising this paper for final publication. In particular, our completion of a CVS repository conversion tool made it possible to perform the first storage overhead measurements for the system design presented in the preceding section. Test cases that run nicely on 10 or 15 operations do not reveal problems that arise when thousands of operations are performed.

Several changes were needed to overcome storage and usability issues revealed by these larger tests. The revised system is now being called OpenCM.

7.1 RCS Storage Revisited

Our first test case was to check in the entire change history of the “build” subdirectory of the EROS tree. The CVS tree for this subsystem occupies 164 blocks (96 if gzipped).² CPCMS, using an uncompressed flat file repository, required 5,504 blocks. The rude surprise was that compression did not help substantially. Compressing the files yielded a CPCMS repository of 4,236 blocks – far from the 20% increase over RCS that we had hoped for.

On examination, we discovered that the objects in the repository corresponding to the original RCS files (the Entity and EntityBits records) accounted only for 40% (2,212 blocks) of the uncompressed total. While the CommitInfo records also record state from the RCS files, they are not stored in a directly comparable way. If CommitInfo records are included as “content” then the content portion of the space rises to 48.8% (2,688 blocks).

This was much higher than expected, but the real surprise was that 60% of the space was going to the configuration schema portion of the store. While we expected to see a proportionally large amount of configuration data for this test case – the build subtree is characterized by few new file introductions and many small changes – this was unexpectedly high.

Our first goal was to reduce the space occupied by the Entity and EntityBits portion of the data. To do so, we built a repository using RCS as the underlying storage layer. Each commit introduced two new cryptographic names in the content schema (one for the EntityBits, the other for the CommitInfo). In the RCS implementation, we tag

² All block numbers reported here are 1 kilobyte blocks

each version using its cryptographic name, which adds still more overhead to each version (there are now three uncompressable names rather than two). Without compression, the resulting repository occupies 1,104 blocks, but the portion of the repository state corresponding to the content schema now occupies only 300 blocks. Given that the Entity vs. EntityBits split has visible impact on merge performance, we are reluctant to unify these two objects. We expect that the XDFS-based system, when implemented, should recover most of the balance of content storage overhead.

7.2 On the Perils of Cryptography

While the content schema using the RCS storage repository occupies only 300 blocks, the total repository size remained excessive. The marginal 804 blocks are recording information that is not stored by CVS, but they seem like a steep price to pay for the configuration schema part of the store. The culprit in the story initially appeared to be cryptographic naming.

Part of the problem is the schema design. There are four objects in the original configuration schema (Branch, Change, ChangeVec, EntitySet) where only one is really needed. These objects are connected by cryptographic names, each of which adds (after encoding) 32 bytes of uncompressable state to every commit. We were able to quickly eliminate the ChangeVec and EntitySet structures by merging them inline into their containing objects. We also made changes to the management of mutables. Where CPCMS implemented multiple mutable object types, OpenCM implements a single mutable type. Mutables name revision records and revision records name content. Together, these reduced the total overhead to 1060 blocks.

We then noticed a bug in the client code – the client was failing to specify a “predecessor” when performing entity revision on the CommitInfo records. Fixing this bug brought the total RCS repository size for the build tree down to 704 blocks. If gzip compression were incorporated, the total size of this repository would be 348 blocks. This suggests that while a delta-based encoding scheme in the repository is worthwhile, the RCS encoding is relatively inefficient. We would prefer to incorporate an encoding scheme directly into OpenCM in any case, as calling RCS has performance consequences.³

Fortunately, the EROS build subtree proves to be a pessimistic test case. Just the thing to prompt a mad scramble to improve a CM system, but not representative of typical usage. Changes in the build subtree are small, so the relative cost of the configuration data overhead is quite high.

³ One possibility would be to integrate the RCS library implementation from CVS and also use zlib as the file I/O interface. We have not yet had an opportunity to investigate this.

When the revised system is run against the “base” subtree of the EROS repository (the tree containing our kernel source and key applications), a somewhat different picture emerges. The CVS repository for this tree is 28,208 blocks, the OpenCM content schema objects take 47,304 blocks, and the total OpenCM storage is 65,692 blocks. If compressed via gzip, the OpenCM numbers reduce to 17,848 and 23,260 blocks respectively. For comparison, the EROS build tree has seen 119 changes, mostly minor, in the last 5 years. The EROS base tree has seen 3199 changes over the same period.

	build	base
Content Only		
CVS	164 (100%)	28,208 (100%)
CPCMS-flat	2,212 (1,348%)	360,376 (1,277%)
OpenCM	300 (182%)	47,304 (167%)
Content Only, gzipped		
CVS+gz	96 (58%)	12,660 (44%)
CPCMS-flat+gz	1,668 (1,017%)	221,692 (785%)
OpenCM+gz	136 (82%)	17,848 (63%)
Total Storage		
CVS	164 (100%)	28,208 (100%)
CPCMS-flat	5,504 (3,356%)	600,698 (2,129%)
OpenCM	704 (429%)	65,692 (232%)
Total Storage, gzipped		
CVS+gz	96 (58%)	12,660 (44%)
CPCMS-flat+gz	4,236 (2,582%)	336,060 (1,191%)
OpenCM+gz	348 (212%)	23,260 (82%)

Table 1: Summary of storage use. CPCMS figures show flat file repository sizes. OpenCM figures show sizes after RCS and Schema repairs are applied. All percentages are relative to uncompressed gzip.

The space performance results are summarized in Table 1. The current OpenCM repository stops short of re-encoding objects for reasons of robustness – re-encoding would impose a need for transaction support that the current server does not require. We are considering additional schema modifications to further reduce storage consumption.

While there is clearly more work to be done, we consider the degree of compressability on the current RCS-based OpenCM repository extremely promising. For the EROS base tree, the reduction from compression in the original RCS repository was 56%, while the reduction from compression (total storage) for OpenCM was 75%. Since cryptographic names are inherently uncompressable, the compression ratio must arise from two factors:

- The fact that OpenCM uses XML as its storage format.

- The fact that the OpenCM schema still contains considerable redundancy.

As a result of these numbers we plan to integrate compression into the OpenCM storage layer, and use a more efficient delta encoding system. We are more cautious about reducing schema redundancy. One of the (currently untested) potential strengths of the current OpenCM schema is the degree to which information can be reconstructed if an object is lost or damaged. We are concerned that reducing this redundancy would reduce the likelihood of successful recovery.

7.3 Naming for Users Redux

As mentioned in Section 3.4, the original pet name design was not viable. The essential lesson from this design was that human naming of objects in OpenCM needed real attention and a basic redesign. To address these issues, we introduced a directory system for human-management object names.

To facilitate human association, each OpenCM server provides a private directory space for each user. Users can “bind” cryptographic names to human-readable names within this space. For example, the command

```
opencm create project eros
```

proceeds as follows:

1. It creates a new mutable object that will represent the identity of this project.
2. It creates a new, frozen project object carrying the initial project description, and sets the “value” slot of the project mutable to point to this object.
3. It creates a new mutable object to serve as the identity of the “eros” directory.
4. It creates a new frozen directory object containing the pair (project, *pm-name*), where *pm-name* is the true name of the project mutable object. This directory object is made the “value” of the directory mutable.
5. It locates the mutable that represents the user’s “home directory”, and revises the content (a directory object) of that mutable to include a new binding (eros, *em-name*), where *em-name* is the mutable name of the previously created eros directory.
6. It revises the user’s top-level directory mutable to point to the new directory object containing the combination of the previous state and the new directory binding for “eros.”

The net effect of all this is that the user can now type:

```
opencm ls
opencm ls eros
opencm show eros/project
```

and obtain a “directory” listing showing respectively that eros is a directory, that this directory contains the name project, and showing the content of the eros/project object.

The new naming system carries an additional benefit: the cvs tag command can now be achieved by creating a duplicate mutable naming the content of some existing branch. Changes to the original mutable can proceed, but the duplicate’s state is unchanged. Mutables can be marked “frozen,” in which case the tagged version can be deleted but not inadvertently altered. Conversely, an unfrozen mutable of this type provides all of the mechanism needed to perform a “branch” operation. The pcms branch command operates in just this way.

In the command line OpenCM client, it proves that this naming scheme extends naturally into the managed object trees. Branch objects are displayed by the opencm ls command as a directory of versions. A given version is in turn displayed as a directory containing the top level names of the managed content. In effect, an entire source tree can be browsed as a single hierarchical namespace. We plan to use this to provide a simple CGI script that displays OpenCM managed content via a web browser.

8 Early Experience

OpenCM is now self-hosting, and we have started using OpenCM internally to support the EROS operating system project. While the development groups in question are small, these groups make heavy use of laptop environments, and therefore routinely test the disconnected operation features of OpenCM.

In the EROS and OpenCM projects, we have a steady supply of captive early adopters (students) who are initially unfamiliar with either OpenCM or EROS. While many of their modifications are ultimately accepted into the main tree, there is some desire for quality control in the acceptance process. This is enforced by encouraging the students to create private branches. While they lack the authority to modify the public development trees, this does not deprive them of the ability to modify, save, undo, branch, or evolve their own line of work. This work can later be (repeatedly) integrated, preserving not just the changes but the history trail and commentary that accompanied them. No comparable mechanism exists in other CM systems we are aware of.

Three differences relative to CVS have been immediately noticeable. The first is that *rename* works without loss of

evolution history. This is rather like having a new tooth implanted: one has grown accustomed to the hole and is surprised on obtaining the implant to realize how much one missed the tooth after all.

The second difference is the relative ease of mobile development. The standard practice is to set up a caching repository on a laptop, create a branch originated by this repository, and check this branch out. As a side effect, the “top” version of the parent branch is copied to the laptop repository. Commit, undo, and history of changes made while on the road is ready to hand. On return from travel, the local repository is synchronized to the main repository and the final configuration (which includes its complete change history) is merged into the parent branch. With only slight modifications to the current design, it will be possible to cache the entire change history text without caching all of the object versions involved.

The third difference is the relative ease of change integration. Instead of mailing a patch, the merge hand-off in OpenCM is accomplished by synchronizing repositories and mailing the name of the branch containing the desired changes. Because the integrator has an entire, connected change graph to work with, the merge is frequently automated. When conflicts arise, the change history of both lines of development is available for examination to decide what to do.

9 Related Work

A number of related efforts exist or are currently underway.

9.1 RCS and SCCS

RCS and SCCS provide file versioning and branching for individual files. The two differ slightly in feature set, and significantly in their storage strategies. Older SCCS implementations are dramatically slower than RCS. The GNU implementation (CSSC) is considerably faster. GNU CSSC uses a storage strategy that can extract arbitrary versions in linear time, where RCS must internally reconstruct various intermediate versions in branching cases.

Both SCCS and RCS provide operation on a single file. Neither provides configuration management.

While either SCCS/CSSC or RCS could be used as an underlying storage implementation for OpenCM, some form of name translation strategy would be required to map true names into SCCS or RCS version names. In the quick and dirty OpenCM RCS repository, we accomplished this by symbolic linking each RCS file under all truenames and tagging each RCS version with its associated truename. The RCS file is treated as a flat, unbranching sequence of

changes. A more deliberate server implementation would accomplish the same mapping using a DB file. One advantage to our choice of encoding robustness: the symbolic links can, if necessary, be recovered from the tag names in the RCS file.

9.2 NUCM

NUCM (University of Toronto) [dHHW96] uses a server information architecture that is similar to that of OpenCM. NUCM “atoms” correspond roughly to OpenCM frozen objects, but atoms cannot reference other objects within the NUCM store. NUCM collections play a similar role to OpenCM mutables, but the analogy is not exact: all NUCM collections are mutable objects. Further, the NUCM information architecture includes a notion of “attributes” that can be associated with atoms or collections. These attributes can be modified independent of their associated object, which effectively renders every object in the repository mutable.

The distinction in their respective handling of collections and mutables is a significant architectural gap between the two systems. The NUCM repository does not have a simple means of providing integrity controls, and the “everything is mutable” design imposes a hierarchical and strongly connected structure on the repositories.

Finally, NUCM versions atoms rather than mutable collections. This is unfortunate, as it conflates the workspace name of the object with its content, and requires that the repository impose a canonicalization policy on names.

While we were initially attracted to the NUCM server as a possible base for OpenCM, we discovered on reading the repository code that it has not been written with robustness in mind. System call return codes were not checked in the version we examined (in fairness, this may since have been fixed). CM systems are integrity-critical systems. We cannot say from experience that NUCM is unreliable, but given the lack of error checking in the code we are unwilling to commit a large enough work base to NUCM to find out.

9.3 BitKeeper

BitKeeper [McV] is similar in feature set to OpenCM, in that it provides multiple satellite repositories and change replication. BitKeeper does not provide cryptographic integrity controls, and its authentication model is not based on cryptographic authentication. Provenance tracking in BitKeeper across multiple organizations is unreliable in the absence of a universal authentication system. In the absence of integrity protection, hostile replicates can inject modified copies of code in such a way that clients cannot detect the substitution.

A secondary concern about BitKeeper is licensing. While the “free if you use our log server” license is appealing, a license that straddles the free/pay boundary is difficult to manage in projects that span commercial and public project members. Subjectively, it seems clear that BitKeeper has failed the test of user acceptance in the open source community. For many users this is unimportant. Given that the core of our own work is open source we feel that this is a significant concern for our applications.

9.4 CM Systems

A brief examination of existing tools is what convinced us that a new CM system was needed. Commercial CM tools did not provide distribution. *Subversion*, the successor to CVS, had elected to defer the question of distribution and adopted (in our view) an information architecture that was unlikely to distribute easily. It also requires long transactions that are relatively easily disrupted by loss of a phone line.

Given the statements of Section 6, it is hopefully clear that PRCS [MHS98] significantly influenced our design. The OpenCM merge strategy is directly borrowed from PRCS, and we are very grateful to Josh MacDonald for the time he put in discussing replication options with us. As with OpenCM, future versions of PRCS are expected to be built on XDFS. In contrast to OpenCM, PRCS provides client/server connection but not distribution. Discussions with Josh MacDonald at the time the OpenCM project was started suggested that distribution would not be available in our timeframe, which encouraged us to pursue OpenCM quasi-independently.

9.5 WebDAV

The “Web Documents and Versioning” [WG] initiative is intended to provide integrated document versioning to the web. It provides branching, versioning, and integration of multiple versions of a single file. When the OpenCM project started, WebDAV provided no mechanism for managing configurations, though several proposals were being evaluated. Given the current function of OpenCM, OpenCM could be used as an implementation vehicle for WebDAV.

9.6 Other

Both Microsoft’s “Globally Unique Identifiers” and Lotus Notes object identifiers are generated using strong random number generators. Mark Miller’s capability-secure scripting language *E* [MMF00, Mil] uses strong random numbers as the basis for secure object capabilities. The *Droplets* system by Tyler Close has adapted this idea to cryptographic capabilities encoded in URLs.

The Xanadu project was probably the first system to make a strong distinction between mutable and frozen objects (they referred to them respectively as “works” and “editions”) and leverage this distinction as a basis for replication [SMTH91]. In hindsight, the information architecture of OpenCM draws much more heavily from Xanadu ideas than was initially apparent.

10 Acknowledgments

Mark Miller first introduced us to cryptographic hashes, and assisted us in brainstorming about their use as a distributed naming system. Josh MacDonald took time for a lengthy discussion of OpenCM and PRCS in which the strengths and weaknesses of both were exposed. It can fairly be said that this conversation provided the last conceptual validation of the idea and prompted us to go ahead with the project. Various discussions on the Subversion mailing list pointed out issues we might otherwise have failed to consider. Paul Karger first pointed out to us the requirement for traceability in building certifiable software systems, and the fact that given then-current tools this requirement was incompatible with open-source development practices.

Within the Hopkins Systems Research Laboratory, Raphael Schweber-Koren split the original monolithic implementation into client/server form, implemented the server connection and dispatch loop and the client side session cache.

Niels Provos served as the shepherd for this paper. His suggestions and comments have significantly improved the paper as you see it. While insisting on clarity and accuracy, Niels simultaneously maintained a constructive and supportive tone throughout our exchange. This balance, in our experience, is difficult to achieve and maintain. We appreciate his efforts on both counts.

11 Future Work

The OpenCM command line client was an obvious first step, but we are considering two possible directions for enhancement on the client side. The first is to build a GUI-based client, enabling the user to get a continuously monitored overview of their project status. Comparable front-ends have been built for CVS, but for OpenCM, we feel that a complete, separate client is potentially appropriate. Indeed, there is no fundamental reason why the visual and command line clients cannot be used at the same time by the same user if appropriate locking disciplines are observed in the workspace. The second is IDE integration, allowing OpenCM to be directly used in various development environments.

A second direction for further work is repository storage.

As previously mentioned, the XDfs versioned file system is a nearly ideal substrate for use as a OpenCM store. Integrating this will significantly reduce the server-side storage overhead of OpenCM.

There is a problem in the current design concerning scope of potential theft. The client-side file that records the workspace state contains the true name of the configuration object from which the workspace was constructed. This object in turn has predecessor names that (transitively) name the entire history of the project. Frozen objects are not uniquely associated with projects or branches, and frozen object fetches therefore are not individually access checked by the repository in the current design. Any user who can authenticate to the server and present a valid true name – even if they can only authenticate as the anonymous user – can obtain a frozen object.

Ryan Goltry, a student at Hopkins, has proposed a modification to OpenCM in which every user of OpenCM has a unique encryption key that is used to secure and validate their entity names. In this design, the entity names stored on the client would be encrypted in a user-specific way. If stolen, they are useless without the user's pass phrase, and the user's encryption key can be invalidated on the server without significant loss if necessary. One beauty of public key cryptography is that the client-side workspace file in this scenario can be re-encrypted without losing any changes that may be in progress.

12 Conclusion

OpenCM provides first-class support for configurations, support for disconnected commit and distributed repositories, per-file access controls, and support for development groups that span multiple companies or administrative domains.

A recent discussion on the linux-kernel mailing list [Bro02] generating the following (edited) list of desirable CM system features. We have annotated each to indicate which ones are addressed by OpenCM:

1. Working merges [yes]
2. Atomic checkins of entire patches, fast tags [yes]
3. Graphical 2-way merging tool. [no]

This is a very important aspect of a successful CM client that we have not yet addressed. A graphical merge mechanism could very easily be integrated into OpenCM, and we would be happy to adopt a reasonable revision from the community to support it.

4. Distributed repositories [yes]

5. Ability to exchange changesets by email [yes*]

OpenCM goes one better – email an OpenCM URI that directly references the new configuration on the developer's server. This preserves not just the changes, but the *history* of the changes.

6. Ability to rename files [yes]
7. Ability to do archival and renaming of directories. [yes]
8. Remote branch repository support [yes]
9. Support for archiving symlinks, device special files, fifos, etc. [no]

Version 0.1 of the OpenCM client incorporates type tags in entities, but does not currently know how to interpret any type other than file. Addition of new entity types is straightforward, and we would be happy to adopt a reasonable revision from the community.

While the linux kernel effort is an extreme test of any CM system, we suspect that these features will also be useful to other projects.

OpenCM is built on a small set of simple ideas that are pervasively applied. While there are many interdependencies in the design, there are no clever or excessively complicated algorithms or techniques in the system. The fundamental insight, such as it is, is that successful distribution and configuration management can be built on only two primitive concepts – naming and identity – and that cryptographic hashes provide an elegant means to unify these concepts.

The core OpenCM system, including command line client, local flat file repository, RCS repository, and remoting SSL support, consists of 18,896 lines of code. In contrast, the corresponding CVS core is over 62,000 lines (both sets of numbers omit the diff/merge library). In spite of this simplicity, OpenCM works reliably, efficiently, and effectively. It also provides greater functionality and performance than its predecessor. One of the significant surprises in this effort has been the degree to which a straightforward, naïve implementation has proven to be reasonably efficient.

OpenCM is available for download from the EROS project web site (<http://www.eros-os.org>) or the OpenCM site (<http://www.opencm.org>). A conversion tool for existing CVS repositories is part of the distribution.

References

- [Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.
- [Bro02] Zack Brown. Kernel traffic #158 for 18 mar, 2002. http://kt.zork.net/kernel-traffic/kt20020318_158.html.
- [dHHW96] A. Van der Hoek, D. Heimbigner, and A. Wolf. A generic peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering*, Berlin, Germany, March 1996.
- [FIP94] Security requirements for cryptographic modules. In *Federal Information Processing Standards Publication 140-1*, 1994.
- [Mac00] J. MacDonald. File system support for delta compression, 2000.
- [McV] Larry McVoy. BitKeeper web site. www.bitkeeper.com.
- [MHS98] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. In *System Configuration Management*, pages 33–45, 1998.
- [Mil] Mark S. Miller. The E web site. www.erights.org.
- [MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proc. Financial Cryptography 2000*, Anguila, BWI, 2000. Springer-Verlag.
- [SFH⁺99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.
- [SMTH91] Jonathan S. Shapiro, Mark Miller, Dean Tribble, and Chris Hibbert. *The Xanadu Developer's Guide*. Palo Alto, CA, USA, 1991.
- [SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.
- [SVN] Tigris.Org. *Subversion Web Site*. <http://subversion.tigris.org>.
- [Tic85] Walter F. Tichy. RCS: A system for version control. *Software – Practice and Experience*, 15(7):637–654, 1985.
- [WG] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the Sixth European Conf. on Computer Supported Cooperative Work (EC-SCW'99)*, Copenhagen, Denmark, September 12-16, 1999, pages 291–310.

X Meets Z: Verifying Correctness In The Presence Of POSIX Threads

Bart Massey

*Computer Science Department
Portland State University
Portland, Oregon USA 97207-0751
bart@cs.pdx.edu*

Robert Bauer

*Rational Software Corporation
1920 Amberglen Pkwy, Suite 200
Beaverton, Oregon USA 97006
rbauer@rational.com*

Abstract

The engineering of freely-available UNIX software normally utilizes an informal analysis and design process coupled with extensive user testing. While this approach is often appropriate, there are situations for which it produces less-than-stellar results.

A case study is given of such a situation that arose during the design and implementation of a thread-safe library for interaction with the X Window System. XCB is a C binding library for the X protocol designed to work transparently with either single-threaded or multi-threaded clients. Managing XCB client thread access to the X server while honoring the constraints of both XCB and the X server is thus a delicate matter.

The problem of ensuring that this complex system is coded correctly can be attacked through the use of lightweight formal methods. A model is constructed of the troublesome portion of the system using the Z formal specification notation. This model is used to establish important system properties and produce C code with a high likelihood of correctness.

1 Introduction

The design and implementation of multi-process and multi-threaded systems has historically been fraught with peril. This sort of code is commonly defective in subtle ways, leading to errors that are difficult to reproduce and troublesome to diagnose.

Each of the past several years, one of the authors has taught a course on formal methods for modeling and analysis of software systems using the Z (pronounced "Zehd") specification notation. This course has reviewed numerous successful case studies of "lightweight" formal methods using Z and related notations to analyze and formalize requirements and design

and guide the development of resulting code. While this approach is not a panacea, it typically does dramatically reduce the likelihood of defects and increase confidence in the developed system. A more surprising finding of most of these case studies, confirmed by the authors' experience in teaching formal methods to industrial practitioners, is that Z-like lightweight formal methods are quite accessible to reasonably experienced software engineers. In fact, in many studies, the savings in time and effort due to reduced defect rates has exceeded the extra cost of judiciously-applied formal analysis.

Thus, using the Z notation to construct a lightweight problem model can enable the construction of an algorithm for controlling client thread access to the server that has a high likelihood of correctness in a situation where user testing is likely to be ineffective. The methods employed, while challenging, can be learned with a reasonable amount of effort by experienced programmers. This approach is thus a useful adjunct to more traditional methods of freely-available software construction.

2 Background

The X Window System [SGN88] has for some 15 years been the underlying basis common to most graphical user interfaces for UNIX systems. During that time, the server and wire protocol have been generally praised as well-designed. Indeed, the sample server implementation provided originally by the MIT X Consortium and now maintained and enhanced by the XFree86 team is widely regarded as the prime example of a policy-free networked graphics platform.

The client side software, however, has frequently been criticized for its somewhat excessive resource consumption, mediocre performance and reliability, and inflexibility in the face of changing application demands. Such criticisms seem especially well-justified in the case of

the new generation of handheld UNIX platforms that often come with a modest CPU and memory, are mission critical, and feature GUIs constrained by limited screen size and other factors. A number of recent X “toolkits”, for example GTK+ [Pen99] and Qt [Dal01], have eschewed the original X GUI approach involving Xt [AS90] and the Athena or Motif [You90] widget set. This has led to a marked improvement along the critical client-side axes.

Underneath almost all these modernized toolkits, however, communication with the X server is still typically handled by Xlib [SGN88]. The Xlib C binding to the X protocol is in some ways a superior piece of code: its quality is attested to by its longstanding sole ownership of this critical task. However, Xlib suffers, to a lesser degree, to many of the problems that plagued Xt.

The XCB “X C Binding” [MS01] is an attempt to provide a radically shrunken and simplified X protocol translation library for C programs, while simultaneously extending flexibility in a few key ways demanded by modern toolkits and applications. Primary among these extensions is first-class support for multi-threaded library clients. The XCB interface attempts to provide a convenient latency-hiding interface to multi-threaded X clients, while cleanly and efficiently handling single-threaded clients using the same interface.

A principal difficulty in the design of the XCB internals was establishing the correctness of the core routines that handle the transfer of X requests to the server and server replies and events back to the client. The complication here is that these routines must interlock properly when being invoked by several threads, and must also complete successfully in the single-threaded case. Reasoning about multi-threaded code is always difficult. The IEEE POSIX 1003.1c-1995 Threads Specification (PTHREADS) [IEE95] provides details of the required semantics of the thread system itself (although this is difficult to verify, since the cost of the printed specification is prohibitive and it is not readily available online or through the local library). The trick is to avoid the perils of deadlock, races, non-determinism, and spinning that seem to be inherent in real multi-threaded situations.

The Z specification notation [Spi92] is a specific syntax and semantics for first-order logic with finite and integer sorts. It is carefully designed to avoid logical paradoxes, to be easy to use to describe realistic software engineering problems and solutions, and to be amenable to pencil-and-paper proofs of important properties. There are many excellent books available to introduce Z to software engineers (*e.g.*, [Jac97, Wor92]), and the existing knowledge typical of software engineers greatly assists learning Z.

During the construction of XCB, it became apparent that an important component, the XCB_Connection layer, is quite difficult to make correct while simultaneously accommodating the external design constraints placed upon it. Essentially, a “thread scheduling” algorithm with somewhat unusual requirements is needed. Four different designs for this algorithm were pseudo-coded, and two of these designs were actually implemented in C. Each of these four designs was subsequently shown, after a great deal of thought, to be subtly incorrect.

The task of specifying the thread scheduling algorithm using Z was begun with the hope that a reasonable amount of work could produce a design that was arguably, if not provably, correct. This hope has proven out: the Z model described here specifies such an algorithm.

It is somewhat unusual in the literature to present a Z model for a real system in its entirety, especially to an audience not necessarily comfortable with the Z notation. Nonetheless, it is hoped that this full presentation will provide a gentle introduction to Z, illustrate some interesting details of formal modeling on a real-world problem, and show how formal methods can be of assistance to developers of freely-available software for UNIX systems.

3 A Z XCB_Connection Model

A precise description of the XCB_Connection layer of XCB is needed in order to prove the desired properties. This layer of XCB handles calls from client threads (through the XCB_Protocol layer) to perform several tasks: enqueueing X server requests, retrieving X server replies, and retrieving X events (generated spontaneously by the server).

The principal engineering difficulties of the XCB_Connection layer are twofold. First, it is desirable for a single API to transparently handle both the single-threaded and multi-threaded case. Second, it is desirable that both of these cases be handled with minimum possible client latency: the layer should never gratuitously block a thread that could continue.

In this section, an engineering solution for the XCB_Connection layer is described. Semi-formal arguments about correctness of this solution are also offered.

3.1 The Z Notation

The style of the description of XCB_Connection presented is that standard for Z specifications: natural-language text interleaved with Z paragraphs. No prior

knowledge of Z is assumed here. Thus, in this section and the succeeding description an attempt will be made to explain the semantics of the various Z constructs used in the description. For Z novices, it may be desirable to consult one of the Z references cited in Section 2 before attempting to read this section.

A Z specification describes a state machine. States in the Z specification correspond to machine states of the program being modeled. State transitions in the specification correspond to execution of the program. In Z, state transitions are expressed using constraints between unprimed variables x indicating the state before the transition and primed variables x' indicating the state after the transition.

A Z state machine is described using *paragraphs* consisting of two optional parts. A declaration section lists the data objects comprising the state and their types. A constraint section indicates, in the form of equational first-order logic, constraints on the data items that must hold for the paragraph. The constraint portion of a Z paragraph limits the permissible values of variables named in the declaration portion of the paragraph. The declared type of a variable also limits its legal set of values. A horizontal bar separates the declaration section from the constraint section when both are present.

Z paragraphs come in several flavors. Basic type definitions define the names associated with types. In Z, a type is synonymous with the set of values that comprise it. Any set of values of a given type can itself be used as a type: thus new types can be constructed from old.

Axiomatic definitions are written with a bar to the left: they define the names and constraints associated with globally visible constant values. Schema definitions are written with a bar to the left, above, and below, forming a “schema box”: they are typically named, and define names and constraints associated with a state or state transition. Since a schema definition denotes a range of possible values for a set of state variables, the name of the schema may itself be used as a type.

Z is not a programming language. As in mathematics, variables are simply names for values: the value associated with a name does not change over the course of a Z description. There is no inherent notion of execution of a Z description: the description defines a state machine whose transitions model the execution of a computer program, but this state machine is not necessarily directly executable.

Nonetheless, a Z description is useful in several ways: it can precisely specify the behavior of a computer program, it can expose logical defects in program design, and it can enable informal or rigorous proofs of desired

design properties. Like other mathematical notations used in engineering, Z is a powerful tool for producing a high-quality product, rather than a substitute for the product itself.

3.2 Modeling System State

An item of principal interest in the description of the XCB system state is the set of client threads using the API. In Z, the existence of a set with no particular defining characteristics is declared by placing the set’s name in square brackets: this is known as a “base type” declaration, since such a set is typically used as the set of values constituting some type. The Z here declares such a set, *THREAD*. The second line of the paragraph declares that *SEQNUM* is another name for \mathbb{N} , since the “sequence numbers” used by the X protocol are natural numbers.

```
[THREAD]
SEQNUM == N
```

3.2.1 X Server State

The X server may be either ready to accept requests or not at any given time. At any given time, it may have a reply or an event response available. The sequence number of a reply is important to its correct processing. The Z here declares the existence of a set *REQ_STATUS* consisting of two distinct elements, given the names *req_ready* and *req_not_ready*. This is known as a “free type” declaration, since the set of values declared is typically used as a type. The declaration of *RESP_STATUS* is similar: in this case, the set consists of distinct elements named *no_resp* and *event*, and a set of distinct elements produced by applying the implicitly-defined “constructor” function *reply* to any legal *SEQNUM*.

```
REQ_STATUS ::=
    req_ready | req_not_ready
RESP_STATUS ::=
    no_resp | reply(⟨SEQNUM⟩) | event
REPLY == ran reply
```

The X server state is described by a record containing the request and response status. The Z *schema* for this describes something analogous to a two-field record, where the *req_status* and *resp_status* fields, above the line in the Z “schema box”, have the given types.

```
ServerState
req_status : REQ_STATUS
resp_status : RESP_STATUS
```

3.2.2 XCB Client State

The state of the XCB client application is much easier to describe. Its only interesting property, for the purposes of this description, is the set of threads that are blocked waiting for XCB. The possible sets of blocked threads are described as elements of the *power set* of threads (*i.e.*, the set of all possible sets of threads).

ThreadState
blocked : $\mathbb{P} \text{ THREAD}$

3.2.3 XCB_Connection State

Having described the state of the external entities, it is necessary to describe the state of XCB_Connection itself. This state is a function of the client calls received, as well as the external states seen.

Important properties of an XCB client call, from the point of view of XCB_Connection, are the sequence number of interest and the thread making the call.

AskInfo
seqnum : *SEQNUM*
thread : *THREAD*

An XCB client call is referred to in this description as an “ask”. There are three kinds of asks: read asks, write asks, and event read asks. Both read and write asks have a particular sequence number and thread associated with them. (The sequence number is assumed to be generated by the caller; this is not how the system actually works, but it is common in Z modeling to abstract away such details to simplify the treatment.) The sets of read asks and write asks will be used extensively as types, so abbreviations are created for them. Note that these subsets of *ASK* are simply the ranges of their constructor functions.

ASK ::=
 read_ask⟨⟨*AskInfo*⟩⟩ |
 write_ask⟨⟨*AskInfo*⟩⟩ |
 event_ask
READ_ASK == ran *read_ask*
WRITE_ASK == ran *write_ask*

At any given time, there may be one thread that XCB_Connection designates as a “worker”. The worker thread is blocked trying to process (send to or receive from the X server) the request or response associated with a particular sequence number. It is easiest to model

- Reader queue, containing threads waiting to
 - Get an event.
 - Get a reply to an X request.
- Writer queue, containing threads waiting to write an X request.
- Event queue, containing events waiting to be delivered.
- Reply queue, containing replies waiting to be claimed.
- An output FIFO buffer.
- Indication as to whether there is currently
 - A current reader (thread blocked reading).
 - A current writer (thread blocked writing).

Figure 1: Relevant XCB Data Structures

this in Z by having the worker be an ASK, capturing just the data needed in the description.

WORKER ::=
 no_worker | *worker_is*⟨⟨*ASK*⟩⟩

The XCB state is fairly complicated. Figure 1 shows a sketch of the key items as abstracted from pseudo-code constructed during the development process. The state contains a sequence of asks enqueued as writers (a sequence since writes must occur in the order in which the asks are received by XCB), a set of asks enqueued as readers, and a set of replies waiting for a read ask. The queue of events waiting for an event read ask is modeled by a simple counter denoting its cardinality, since the content of events is not important to the description. Finally, the current workers blocked on reading and writing are maintained by the system.

XCBStateVars
writers : seq *WRITE_ASK*
readers : $\mathbb{P} \text{ READ_ASK}$
replies : $\mathbb{P} \text{ REPLY}$
nevents : \mathbb{N}
cur_reader, *cur_writer* : *WORKER*

3.3 Modeling System State Change

The state changes in the XCB_Connection layer are complex. The pseudo-code algorithms of Figures 2 and 3 informally describe the intended state changes of the system. A “reader” is an XCB client thread attempting to retrieve an X server event or a reply to an X server request. (These cases are handled nearly identically by the implementation.) A “writer” is an XCB client thread attempting to send an X server request.

Note that an attempt was made to present the pseudo-code of Figures 2 and 3 as similarly as possible to its appearance in the design notes. This is necessary to ensure that the Z is developed to describe the desired algorithm, rather than driving some completely different algorithm design. It also gives some insight into the algorithm design style used by the XCB authors.

The complexity here is largely due to the restrictions the X protocol specification places on clients. X clients must never assume that the server connection will eventually become writable unless it is continuously being read from. This is because the X server can spontaneously generate responses on its output (X events) that will block the server until they are read. If the client is blocked waiting for the server to read its request, deadlock will result.

This restriction on the protocol is a difficult fit to one of the XCB requirements: that single-threaded and multi-threaded applications be able to use the same interface for sending requests. The single-threaded case by itself would be easy: a client thread wishing to send a request could use the `select()` system call to wait until the socket is writable, in which case it sends a request, or readable, in which case it processes whatever response has appeared and calls `select()` again. With an interface designed for multi-threaded applications, it would be possible to require the XCB client to always have a thread blocked reading events.

With a possibly multi-threaded client, the situation is more complex. The problem arises as follows: while one thread is blocked waiting for an X server response (selecting only for reading), another thread arrives hoping to send a request to the server. The second thread must block until the server connection is writable, but cannot guarantee that the reader will continue to read from the socket: the reader may complete its work and exit. This causes a danger of deadlock: if there is no thread consuming server responses, the server may block, which will block the writer. The solution is for writers to `select()` for both reading and writing, as in the single-threaded case, and handle the resulting race separately.

```
pending:
  Can satisfy call
  from pending queue?
  Yes:
    done
  No:
    Is there a reader or writer?
    Yes:
      enqueue self
      block
      go to pending
    No:
      become reader
select:
  select on read
  is there a writer?
  Yes:
    enqueue self
    block
    goto pending
  get response
  is it ours?
  Yes:
    wake up top of queue
    (writer else reader)
    done
  No:
    Is a client waiting?
    Yes: deliver to it
    No: place in pending
  Is a writer queued?
  Yes:
    enqueue self
    switch to writer
    block
    go to pending
  No:
    go to select
```

Figure 2: XCB Reader Pseudo-code

Z is nice for modeling the XCB pseudo-code, because states are modeled as changing instantaneously: this avoids any explicit mention of mutual exclusion via mutexes at the model level. (Indeed, the presented pseudo-code does not mention them.)

A useful building block in constructing the Z descriptions will be a function that returns the reply in the reply queue matching a given ask, if any. The `OPT_REPLY` type is used to indicate whether there was a reply, and if so, what it was.

```
OPT_REPLY ::=
  no_reply | reply_is(⟨REPLY⟩)
```

```

Is there a writer?
Yes:
    enqueue self
    block
become writer
select:
    select on read  $\vee$  write
    read:
        get reply
        Is there a waiting client?
        Yes: deliver to client
        No: place in reply or
            event queue
        go to select
    write:
        try to write
        was full amount?
        Yes: done
        No: go to select

```

Figure 3: XCB Writer Pseudo-code

The *lookup_reply* function looks up the reply matching the given *ASK* in the reply queue. Note the use of Z *schema inclusion* here: in the first line of the schema, all of the names declared in *XCBStateVars* are included in *XCBState* as well.

The definition of *lookup_reply* is given by constraining the result over every possible *AskInfo*. This style of implicit definition is normal in Z. The *if – then – else* is not an imperative programming-language construct: it is more like the *?:* operator of C, an operator that returns one of two different values depending on its leftmost argument.

```

XCBState
XCBStateVars
lookup_reply : READ_ASK  $\rightarrow$  OPT_REPLY

 $\forall r : \text{AskInfo} \bullet$ 
    lookup_reply (read_ask r) =
        if (reply r.seqnum)  $\in$  replies
        then reply_is (reply r.seqnum)
        else no_reply

```

The constraint, below the line in the Z schema box, defines the *lookup_reply* function. This constraint is important in the proofs: it is implicit in any reference to *XCBState* in subsequent Z in the description. It is this feature of Z that makes it a useful pencil-and-paper proof notation: by expanding schemata (replacing their names with their definition) and manipulating the resulting constraints, one can construct proofs of interesting properties using first-order logic.

This completes the model of the XCB environment to the level of detail necessary for what follows. The formal notation here has already accomplished several tasks relative to the informal model of Figure 1. First, the formal model is quite precise: it tells exactly what is and is not of interest. Second, the formal model is fully strongly typed. This contributes to the precision (since, for example, sequences are typed differently than sets). It also reduces the likelihood of error: this Z, like most Z published these days, has been checked for correct syntax and type safety by an automated tool. As with computer programs, semantic defects in formal notation are often accompanied by syntactic or type errors.

3.3.1 Read Ask Model

Having described the environment and the state, the next step in Z modeling is to describe the possible state changes during execution. A Z model defines a (not necessarily deterministic) state machine. The initial state of the model is given explicitly. The remaining reachable states are implicitly defined by giving schema denoting legal state transitions. A state transition schema consists of unprimed values (e.g. *x*) of the state before the transition, and primed values (e.g. *x'*) of the state after the transition.

The first three lines of the first state transition schema denote Z schema inclusions. The Δ operator applied to a schema inclusion denotes that the schema should be included twice: once with its names primed and once with them unprimed. Thus, all the definitions and constraints in *XCBState* implicitly appear in *ReadAskQueued* both with and without primed names. This ensures that any invariant constraints defined in *XCBState* hold both before and after the state transition. The Ξ operator includes primed and unprimed schema as *Delta* does. In addition the Ξ operator, that denotes that the referenced state will not change, implicitly defines an equality constraint between each primed/unprimed pair in the schema being operated on. The identifier *ask?* has been decorated with a question mark to indicate that it is an input to the state change, not part of any state.

The *ReadAskQueued* schema starts a read request for *ask?*, by checking whether the reply queue already contains the data being asked for. If so, the resulting reply queue is constrained (by the somewhat complex logic in the unique-existential constraint) to no longer contain the data, and the request is satisfied.

ReadAskQueued

$\Delta XCBState$
 $\Xi ThreadState$
 $\Xi ServerState$
 $ask? : READ_ASK$

$readers' = readers$
 $writers' = writers$
 $\exists_1 r : REPLY \bullet$
 $reply_is\ r = lookup_reply\ ask? \wedge$
 $replies' = replies \setminus \{r\}$
 $nevents' = nevents$
 $cur_reader' = cur_reader$
 $cur_writer' = cur_writer$

What if the data is not available? Then the existential precondition (constraint on the before state) fails to hold. In this case, the entire state transition is unavailable: unless there is some other transition schema that can apply, the state is terminal.

In this case, one transition schema that could apply is one in which the ask blocks waiting for a reply. In the model threads block, not asks, so as a convenience a partial function is defined that returns the thread of a read or write ask. This function is specified by constraining its explicit *map*: the set of input-output pairs that constitute the function.

$ask_thread : ASK \mapsto THREAD$
 $ask_seqnum : ASK \mapsto SEQNUM$

$ask_thread =$
 $\{a : AskInfo \bullet (read_ask\ a \mapsto a.thread)\} \cup$
 $\{a : AskInfo \bullet (write_ask\ a \mapsto a.thread)\}$
 $ask_seqnum =$
 $\{a : AskInfo \bullet (read_ask\ a \mapsto a.seqnum)\} \cup$
 $\{a : AskInfo \bullet (write_ask\ a \mapsto a.seqnum)\}$

The preconditions of the following schema and the previous one are incompatible: the previous schema required that *lookup_reply* return a reply, whereas this one required that it return *no_reply*. Making the preconditions of all state transitions mutually exclusive makes the overall system deterministic. Another precondition of this schema is that the ask's thread cannot itself become the reader, because some other thread is already reading or writing.

ReadAskBlocks

$\Delta XCBState$
 $\Delta ThreadState$
 $\Xi ServerState$
 $ask? : READ_ASK$

$readers' = readers \cup \{ask?\}$
 $writers' = writers$
 $replies' = replies \wedge$
 $no_reply = lookup_reply\ ask?$
 $nevents' = nevents$
 $cur_reader \neq no_worker \vee$
 $cur_writer \neq no_worker$
 $cur_reader' = cur_reader$
 $cur_writer' = cur_writer$
 $blocked' = blocked \cup \{ask_thread\ ask?\}$

Finally, if the reader's data is not yet available and yet no other thread is already prepared to acquire it, this read ask becomes a worker on its own behalf and on behalf of those who come after it.

ReadAskWorker

$\Delta XCBState$
 $\Delta ThreadState$
 $\Xi ServerState$
 $ask? : READ_ASK$

$readers' = readers$
 $writers' = writers$
 $replies' = replies \wedge$
 $no_reply = lookup_reply\ ask?$
 $nevents' = nevents$
 $cur_reader = no_worker \wedge$
 $cur_reader' = worker_is\ ask?$
 $cur_writer' = cur_writer = no_worker$
 $blocked' = blocked \cup \{ask_thread\ ask?\}$

Putting all of this together, if XCB receives a read ask, it should take one of the three state transitions indicated above.

$ReadAsk == ReadAskQueued \vee$
 $ReadAskBlocks \vee ReadAskWorker$

(This shorthand notation for the *ReadAsk* schema indicates that at least one of the three referenced schema always applies: it is possible, though more cumbersome, to write the definition out in full schema notation.)

There are several properties of the specification that can be checked at this point, either informally or using a

formal proof. It is important to verify that the specification is well-founded: that all functions are applied only to their defined domains, for example, and that division by zero never occurs. This check is usually informal, although in this instance some progress was made toward a formal proof using the Z/EVES theorem prover [CMS99].

A well-founded specification may be deterministic or non-deterministic. As discussed earlier, this specification is deliberately deterministic. While non-deterministic specifications are sometimes useful, non-determinism is also commonly an inadvertent result of specification error.

Finally, a well-founded specification will be complete. In this case, it is necessary to show that for any possible combination of XCB states, thread states, and server states, a read ask will meet the precondition of at least one of the given schemata. A brief inspection shows this to be the case here.

3.3.2 Write Ask Model

The action taken by XCB on receiving a write ask is quite similar to that taken on receiving a read ask. One minor difference is that the writes must be enqueued in order: as mentioned earlier, this is the reason that the writers are stored in a sequence. Reads deliver specifically-requested data, and are completed in the order the data becomes available.

Another more important difference from the read case has to do with the situation where the X server is busy and thus writes cannot proceed. In this case, the writing thread cannot simply block until the server is available, since the server may also block waiting for its responses to be consumed, leading to deadlock. Instead, the writing thread must process X server responses while waiting.

Most confusingly of all, if a reader thread is processing X server responses when a write ask comes in and cannot proceed, the writing thread should take over the response processing task after the reader is done processing its next request. Note that there will be a period of time during which both the reader and writer will awaken if data becomes available from the X server. It is the details of this process that caused the most confusion in initial attempts to design this algorithm, and has led to the use of Z to clarify and validate the design.

First, the simple case: if a write ask comes in and there is already a writer, the asking thread blocks until the pending write completes. (In the actual implementation, the fact that there is a write queue and requests are written

in blocks complicates the situation a bit. However, this level of detail is irrelevant to the purpose of modeling.) The blocked ask is used to construct a single-element sequence (surrounded by \langle and \rangle) that is appended (using the sequence concatenation operator \frown) to the queue of pending write asks.

WriteAskBlocks

$$\begin{aligned} &\Delta XCBState \\ &\Delta ThreadState \\ &\exists ServerState \\ &ask? : WRITE_ASK \end{aligned}$$

$$\begin{aligned} readers' &= readers \\ writers' &= writers \frown \langle ask? \rangle \\ replies' &= replies \\ nevents' &= nevents \\ cur_writer &\neq no_worker \\ cur_reader' &= cur_reader \\ cur_writer' &= cur_writer \\ blocked' &= blocked \cup \{ask_thread\ ask?\} \end{aligned}$$

If the write ask can proceed, the thread simply becomes the distinguished writer and blocks waiting to be able to write.

WriteAskWorker

$$\begin{aligned} &\Delta XCBState \\ &\Delta ThreadState \\ &\exists ServerState \\ &ask? : WRITE_ASK \end{aligned}$$

$$\begin{aligned} readers' &= readers \\ writers' &= writers \\ replies' &= replies \\ nevents' &= nevents \\ cur_writer &= no_worker \wedge \\ cur_writer' &= worker_is\ ask? \\ cur_reader' &= cur_reader \\ blocked' &= blocked \cup \{ask_thread\ ask?\} \end{aligned}$$

These schema are then assembled into a whole.

$$WriteAsk == WriteAskBlocks \vee WriteAskWorker$$

Again, questions of validity, determinism, and completeness arise, and again the answers are relatively straightforward.

3.3.3 Server Request Model

When the X server is capable of accepting a request and a writer thread is blocked waiting for this eventuality, the request can be immediately written to the server. At that point, the thread may return from XCB, but not before waking up a pending writer or reader. It is easiest to model this last bit first.

An auxiliary function will be useful for finding the thread associated with a worker. This example shows another popular style of function definition in Z: constraining a function's outputs over all valid inputs. Consider the universally-quantified constraint on *worker_ask*. For every object of type *WORKER* other than *no_worker*, this constraint ensures a valid value for the function. Thus, *worker_ask* is a partial function (denoted by the \leftrightarrow symbol in the type).

$$\begin{array}{l} \text{worker_ask} : \text{WORKER} \leftrightarrow \text{ASK} \\ \text{worker_thread} : \text{WORKER} \leftrightarrow \text{THREAD} \\ \text{worker_seqnum} : \text{WORKER} \leftrightarrow \text{SEQNUM} \\ \hline \forall a : \text{ASK} \bullet \text{worker_ask}(\text{worker_is } a) = a \\ \text{worker_thread} = \text{ask_thread} \circ \text{worker_ask} \\ \text{worker_seqnum} = \text{ask_seqnum} \circ \text{worker_ask} \end{array}$$

The case where there are available writers is then modeled.

$$\begin{array}{l} \text{WakePendingWriter} \\ \hline \Delta \text{XCBState} \\ \exists \text{ThreadState} \\ \exists \text{ServerState} \\ \text{new_worker!} : \text{WORKER} \\ \hline \text{new_worker!} = \text{worker_is}(\text{head writers}) \\ \text{readers}' = \text{readers} \\ \text{writers} \neq \langle \rangle \wedge \\ \text{writers}' = \text{tail writers} \\ \text{replies}' = \text{replies} \\ \text{nevents}' = \text{nevents} \\ \text{cur_writer} = \text{no_worker} \wedge \\ \text{cur_writer}' = \text{new_worker!} \\ \text{cur_reader}' = \text{cur_reader} \end{array}$$

For the current purposes, it is sufficient to nondeterministically select a new reader if there are pending readers but no writers. In practice, it is probably most efficient to select the reader whose reply is expected next.

$$\begin{array}{l} \text{WakePendingReader} \\ \hline \Delta \text{XCBState} \\ \exists \text{ThreadState} \\ \exists \text{ServerState} \\ \text{new_worker!} : \text{WORKER} \\ \hline \exists r : \text{readers} \bullet \text{new_worker!} = \text{worker_is } r \\ \text{readers}' = \text{readers} \setminus \\ \quad \{ \text{worker_ask new_worker!} \} \\ \text{writers}' = \text{writers} = \langle \rangle \\ \text{replies}' = \text{replies} \\ \text{nevents}' = \text{nevents} \\ \text{cur_writer}' = \text{cur_writer} \\ \text{cur_reader} = \text{no_worker} \wedge \\ \text{cur_reader}' = \text{new_worker!} \end{array}$$

Finally, it may be that there is nothing left to do. Z makes stating this case easy.

$$\begin{array}{l} \text{WakePendingNone} \\ \hline \exists \text{XCBState} \\ \exists \text{ThreadState} \\ \exists \text{ServerState} \\ \text{new_worker!} : \text{WORKER} \\ \hline \text{readers} = \emptyset \\ \text{writers} = \langle \rangle \\ \text{new_worker!} = \text{no_worker} \end{array}$$

Putting these together yields

$$\text{WakePending} == \text{WakePendingWriter} \vee \text{WakePendingReader} \vee \text{WakePendingNone}$$

Finally, when the X server is writable and there is a worker waiting to write to it, the worker is awakened, completes the write, replaces itself as necessary, and is done. The server may become blocked as a result of the write: this is left undetermined. The whole series of these “ServerDo” transitions has some common structure that can be conveniently captured with a subschema.

$$\begin{array}{l} \text{ServerWriteStuff} \\ \hline \Delta \text{XCBState} \\ \Delta \text{ThreadState} \\ \Delta \text{ServerState} \\ \hline \text{replies}' = \text{replies} \\ \text{nevents}' = \text{nevents} \\ \text{writers}' = \text{writers} \\ \text{readers}' = \text{readers} \end{array}$$

The subschema together with the manipulations on the state comprise the write process itself.

<i>ServerDoWrite</i>
<i>ServerWriteStuff</i>
$cur_writer \neq no_worker \wedge$
$cur_writer' = no_worker$
$cur_reader' = cur_reader$
$req_status = req_ready$
$resp_status' = resp_status$
$blocked' = blocked \setminus$
$\{worker_thread\ cur_writer\}$

In the model, two state transitions happen sequentially but atomically: first the write, then the wakeups.

ServerRequest == *ServerDoWrite* ; *WakePending*

As before, checking validity and completeness is straightforward. With the exception of the deliberate nondeterminism in server state, the model is also straightforwardly deterministic.

3.3.4 Server Response Model

When the X server has response data to deliver to XCB, at least one of two conditions should hold. The response may be an event, in which case it may be queued at leisure for future examination. The case in which any sort of response is delivered while no thread is blocked waiting to read data is modeled by merely deferring the delivery until a thread is available: this approach requires no mechanism here. If an event is delivered while a thread is ready to read it, this is handled simply by delivering the event. Note that the server read is handled by leaving its new response indeterminate: while there are obviously constraints on what the server could return next, they are outside the scope of the model.

<i>ServerReadEvent</i>
<i>ServerWriteStuff</i>
$\exists ThreadState$
$cur_writer \neq no_worker \vee$
$cur_reader \neq no_worker$
$cur_writer' = cur_writer$
$cur_reader' = cur_reader$
$replies' = replies$
$nevents' = nevents + 1$
$req_status' = req_status$
$resp_status = event$

If the response is not an event, then there should be some thread blocked in XCB waiting to read the response and dispatch it appropriately. This thread will be either the reader or writer worker.

There is some complexity associated with handling the server response that has to do with the interaction between simultaneous reader and writer workers. It is assumed that all threads wake up from `select()` when a response is available. (This assumption seems natural, nothing in the PTHREADS specification seems to contradict it, and it seems to be the behavior under versions of Linux and Solaris tested by the authors.) No assumptions, however, are made about the order in which threads are awakened.

Thus, the read must be handled carefully: whichever worker wakes up first could perform the read, but there would be a danger of it being erroneously re-performed by the second worker. This race is handled by making any writer worker always perform the read. If there is just a reader, it performs the read: if the reply is destined for the blocked thread, it is returned.

<i>ServerReaderThis</i>
$\Delta XCBState$
$\Delta ServerState$
$\Delta ThreadState$
$writers' = writers \wedge$
$readers' = readers \wedge$
$replies' = replies \wedge$
$nevents' = nevents$
$cur_writer' = cur_writer = no_worker$
$\exists_1 s : SEQNUM \bullet$
$worker_seqnum\ cur_reader = s \wedge$
$resp_status = reply\ s$
$cur_reader' = no_worker$
$req_status' = req_status$
$blocked' = blocked \setminus$
$\{worker_thread\ cur_reader\}$

Otherwise, the reply may be dispatched if there is a reader blocked waiting for it, and any current reader worker continues to wait.

ServerReaderOther

 $\Delta XCBState$ $\Delta ServerState$ $\Delta ThreadState$

 $writers' = writers \wedge$ $replies' = replies \wedge$ $nevents' = nevents$ $cur_writer' = cur_writer = no_worker$ $\forall s : SEQNUM \bullet$ $worker_seqnum\ cur_reader \neq s \vee$ $resp_status \neq reply\ s$ $cur_reader' = cur_reader$ $req_status' = req_status$ $\exists a : READ_ASK \bullet$ $a \in readers \wedge$ $resp_status = reply\ (ask_seqnum\ a) \wedge$ $readers' = readers \setminus \{a\} \wedge$ $blocked' = blocked \setminus$ $\{ask_thread\ a\}$

Finally, if there is no reader yet waiting for it, the reply is simply enqueued.

ServerReaderNone

 $\Delta XCBState$ $\Delta ServerState$ $\Xi ThreadState$

 $readers' = readers \wedge$ $writers' = writers \wedge$ $nevents' = nevents$ $cur_writer' = cur_writer = no_worker$ $\forall s : SEQNUM \bullet$ $worker_seqnum\ cur_reader \neq s \vee$ $resp_status \neq reply\ s$ $cur_reader' = cur_reader$ $req_status' = req_status$ $\forall a : READ_ASK \bullet$ $a \notin readers \vee$ $resp_status \neq reply\ (ask_seqnum\ a)$ $resp_status \in REPLY \wedge$ $replies' = replies \cup \{resp_status\}$

If the writer worker is present and wakes up on a read, it can perform the read, process the result, and continue to select. (Actually, there is the potential for a race condition with both a reader and a writer present: if the writer exits before the reader awakens, the writer must first notify the reader that the read has occurred. This portion of the model is omitted here for simplicity: instead the

not-unreasonable assumption is made that the reader will awaken before the writer exits, and these events are handled atomically by the schemata.)

It could be that the X server reply is destined for the current reader worker. In this case, the reply is simply marked as an *ask!* for future handling.

ServerWriterReadWorker

 $\Delta XCBState$ $\Delta ServerState$ $\Xi ThreadState$ $ask! : READ_ASK$

 $writers' = writers \wedge$ $nevents' = nevents$ $cur_writer' = cur_writer \neq no_worker$ $cur_reader' = cur_reader$ $cur_reader = worker_is\ ask!$ $readers' = readers \wedge$ $replies' = replies$ $\exists_1 s : SEQNUM \bullet$ $ask_seqnum\ ask! = s \wedge$ $resp_status = reply\ s$ $req_status' = req_status$

Alternatively, the reply may be destined for a blocked thread in the reader set. In this case, the blocked reader should be removed from the set and treated as a new ask.

ServerWriterReadBlocked

 $\Delta XCBState$ $\Delta ServerState$ $\Delta ThreadState$ $ask! : READ_ASK$

 $writers' = writers \wedge$ $nevents' = nevents$ $cur_writer' = cur_writer \neq no_worker$ $cur_reader' = cur_reader$ $ask! \in readers \wedge$ $readers' = readers \setminus \{ask!\} \wedge$ $replies' = replies \cup \{resp_status\}$ $\exists_1 s : SEQNUM \bullet$ $ask_seqnum\ ask! = s \wedge$ $resp_status = reply\ s$ $req_status' = req_status$ $blocked' = blocked \setminus \{ask_thread\ ask!\}$

In either of these cases, the writer worker will need to awaken a blocked reader worker. The generated output

ask! is used as an input to the *ReadAsk* schema previously defined (via the \gg operator), effectively restarting the state machine at this point.

```
ServerWriterDoRead ==
  (ServerWriterReadWorker  $\vee$ 
   ServerWriterReadBlocked)  $\gg$ 
  ReadAsk
```

Finally, the reply may just need to be enqueued.

<pre>ServerWriterQueueRead ΔXCBState ΔServerState \existsThreadState readers' = readers \wedge writers' = writers \wedge nevents' = nevents cur_writer' = cur_writer \neq no_worker cur_reader' = cur_reader $\forall a : \text{READ_ASK} \bullet$ resp_status \neq reply(ask_seqnum a) \vee cur_reader \neq worker_is a \vee a \notin readers replies' = replies \cup {resp_status} req_status' = req_status</pre>
--

When the server makes a read available with a writer present, it can be processed using one of the schemata just given.

```
ServerWriterRead ==
  ServerWriterDoRead  $\vee$ 
  ServerWriterQueueRead
```

Putting it all together yields a model for X server state changes that deliver a response to XCB.

```
ServerResponse ==
  ServerReadEvent  $\vee$ 
  ServerReaderThis  $\vee$ 
  ServerReaderOther  $\vee$ 
  ServerReaderNone  $\vee$ 
  ServerWriterRead
```

3.4 The Full Model

The final model simply notes that the system state can change by any of the three transitions noted above: *ReadAsk* (Section 3.3.1), *WriteAsk* (Section 3.3.2), or server state change (Sections 3.3.3 and 3.3.4).

```
ServerStateChange ==
  ServerRequest  $\vee$  ServerResponse
```

```
XCBModel ==
  ReadAsk  $\vee$ 
  WriteAsk  $\vee$ 
  ServerStateChange
```

Of course, any state machine needs an initial state. This is specified with a specially-named state schema. Note that the initial server state is unspecified: this is in keeping with the (lack of) server model.

<pre>InitXCBModel XCBState ServerState ThreadState cur_reader = cur_writer = no_worker readers = \emptyset writers = $\langle \rangle$ replies = \emptyset nevents = 0 blocked = \emptyset</pre>

The state machine defined by this Z model describes the desired behavior of XCB_Connection. In the process of describing it, some important progress has been made toward ensuring that the model is well-defined. The model also gives some important hints about how the implementation might be structured. The data structures it suggests are largely straightforward to implement in a conventional programming language. The few non-deterministic transitions of the specification can all be implemented deterministically without sacrificing correctness or performance. In short, the Z model accurately describes an algorithm that is likely to be both correct and reasonable to implement.

4 Analysis

There are a few important theorems that should be stated and proved about the modeled algorithm.

1. It should be shown that the algorithm is deadlock-free: that any thread that enters the system will eventually return from it. To show this, it is necessary to show that XCB will never try to write from the X server in a situation where it cannot read from it: this prevents a particularly pernicious form of deadlock that was actually present for a time in the

initial Xlib implementation [Get01] in which the X server is blocked trying to write to XCB and XCB is blocked trying to write to the X server.

2. It should be shown that that the algorithm meets the guarantee of XCB that threads will exit XCB “as soon as possible”, that is, as soon as their exit conditions are satisfied. This condition is more difficult to state formally: since the statement involves asserting the existence of a sequence of state transitions, the condition cannot be stated directly in first-order logic.

The authors have used the formal model presented here to argue semi-formally that these properties hold: while space considerations preclude presentation of this proof sketch here, the formal model has been extremely helpful in this regard.

Of course, the formal model should correspond to the C code that implements it. One of the weaknesses of the Z notation is that this correspondence is not generally a mechanical one: that is, the C code is not generated by iterative refinement of the formal model as in, for example, the B Method [Abr96]. (The tradeoff here is that the B Method is substantially more difficult to learn and use.)

The C code in XCB that implements the Z model described here is the result of modifying earlier, flawed code to conform to the model. Less than 100 lines of C code are directly involved in implementing the model: this greatly simplifies the task of keeping the code straight. The XCB code implementing the model is freely available: the authors plan to comment the code with correspondences to the model, but the code as it stands is far from opaque in this regard.

To illustrate the sort of problem that the Z model helps to detect, it is useful to look at Figure 4, a defective section of pseudo-code developed prior to the Z modeling process. In this earlier design, there is only a single worker: writers block until reads complete, and themselves select only for writing. In this situation, the informal proof of property (1) above does not hold: when the worker is waiting to write to the X server, no thread is available to read from the X server, and there is a potential for deadlock. While this defect was discovered through informal reasoning about the system, initial attempts to repair it led to other designs with subtle flaws. This was a principal motivator for Z modeling effort: the resulting proof sketches give confidence that defects are being eliminated rather than just pushed around.

```
...
become worker
select:
  select on write
  write:
    try to write
    was full amount?
    Yes: done
    No: go to select
```

Figure 4: Defective XCB Writer Pseudo-code

5 Evaluation

From the authors’ point of view, the effort described in the previous section has been a successful one. The model, while challenging, has not been an unreasonable amount of work to develop. The transition from model to code is an easy one.

The confidence in the algorithm and implementation gained through the formal modeling process is important. The sorts of defects inherent in earlier incorrect versions of this algorithm would be difficult to isolate through even extensive user testing: these defects tend to be infrequent, hard to reproduce, and hard to understand through examination of the implementation. As a consequence, these defects are difficult to debug and repair: often, large pieces of software infrastructure must be torn down and rebuilt.

A number of industrial software engineers participating in the Oregon Master of Software Engineering (OMSE) Program (<http://www.omse.org>) have been taught the Z formal notation during a ten-week course in modeling and analysis of software systems. While these students are quite bright and hard-working, their success suggests that a working knowledge of formal methods is not impossible for those outside the ivory tower to acquire and appreciate. The sort of lightweight modeling described in the previous section, that isolates a troublesome portion of the system for more detailed study, is especially appropriate in this regard. OMSE students have often observed that the success of these methods for them is largely in the modeling stage: detailed formal proofs may be impossible for them, but they are also rarely necessary.

The world of software development, and especially freely-available software development, is changing. Demand for software continues to grow, and minimum acceptable quality levels are increasing: traditional freely-available software development methods may not be efficient enough to meet these countervailing challenges.

In vast commercial organizations, large amounts of structured unit, integration, and system testing effort can help to meet the quality demands. For individual project developers with limited resources, a more “back-of-the-envelope” approach may be suitable. In most engineering disciplines, use of often relatively unsophisticated mathematical methods has been a key to higher quality without substantial loss of productivity. Perhaps a similar result can be attained in the engineering of software.

Availability

The XCB implementation is freely available under an MIT-style license at <http://xcb.cs.pdx.edu>. The LaTeX source for this document, including the Z model, is also available at this location for interested researchers.

Acknowledgements

Thanks to Clem Cole for continuing to shepherd chronically-late authors through a tough paper. Thanks to Keith Packard for inspiring, assisting, evaluating, and suggesting solutions throughout the XCB development process. Thanks to Jamey Sharp, who implemented XCB and provided an incisive critique of the broken bits. Thanks to Jonathan Wistar for insightful comments on short notice.

References

- [Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs To Meanings*. Cambridge University Press, 1996.
- [AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit—The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.
- [CMS99] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z specifications with Z/EVES. In J.P. Bowen and M.G. Hinchey, editors, *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, 1999.
- [Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, 2001.
- [Get01] Jim Gettys, 2001. Personal communication.
- [IEE95] IEEE 1003.1c-1995: Information technology—interface (POSIX) – system application program interface (API) amendment 2: Threads extension (C language), 1995.
- [Jac97] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.
- [MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.
- [Pen99] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing, 1999.
- [SGN88] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System: C Library and Protocol Reference*. Digital Press, Bedford, MA, 1988.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Sciences. Prentice-Hall, London, second edition, 1992. Freely available online at <http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.ps.gz>.
- [Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.
- [You90] Douglas A. Young. *The X Window System - Programming and Applications with X (OSF-Motif Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1990.

Planned Extensions to the Linux Ext2/Ext3 Filesystem

Theodore Y. Ts'o

International Business Machines Corporation

theotso@us.ibm.com, <http://www.thunk.org/tytso>

Stephen Tweedie

Red Hat

sct@redhat.com

Abstract

The ext2 filesystem was designed with the goal of expandability while maintaining compatibility. This paper describes ways in which advanced filesystem features can be added to the ext2 filesystem while retaining forwards and backwards compatibility as much as possible. Some of the filesystem extensions that are discussed include directory indexing, online resizing, an expanded inode, extended attributes and access control lists support, extensible inode tables, extent maps, and preallocation.

1 Introduction

Linux's second extended filesystem[1] (also known as ext2) was first introduced into the Linux kernel in January, 1993. At the time, it was a significant improvement over the previous filesystems used in the 0.97 and earlier kernels, the Minix and the "Extended" or (ext) filesystem. Fundamentally, the design of the ext2 filesystem is very similar to that of the BSD Fast Filesystem[2].

The ext2 filesystem is divided into *block groups* which are essentially identical to the FFS's cylinder group; each block group contains a copy of the superblock, allocation bitmaps, part of a fixed, statically allocated inode table, and data blocks which can be allocated for directories or files. Like most classic Unix filesystems, ext2/3 uses direct, indirect, double indirect, and triple indirect blocks to map logical block numbers to physical block numbers. Ext2's directory format is also essentially identical to traditional Unix filesystems in that a simple linked list data structure is used to store directory entries.

Over the years, various improvements have been added to the ext2 filesystem. This has been facilitated by a number of superblock fields that were added to the ext2 filesystem just before Linux 2.0 was released. The most important of these fields, the compatibility bitmaps, enable new features to be added to the filesystem safely. There are three such compatibility bitmaps: read-write, read-only, and incompat. A kernel will mount a filesystem that has a bit in the read-write compatibility bitmask that it doesn't understand. However, an unknown bit in the read-only compatibility bitmap cause the kernel to only be willing to mount the filesystem read-only, and the kernel will refuse to mount in any way a filesystem with an unknown bit in the incompat bitmask. These bitmaps have allowed the ext2 filesystem to evolve in very clean fashion.

Today, more developers than ever have expressed interest in working on the ext2/3 filesystem, and have wanted to add or integrate various new exciting features. Some of these features include: preallocation, journaling, extended attributes and access control lists, on-line resizing, tail-merging, and compression. Some of these features have yet to be merged into the mainline ext2 code base, or are only available in prototype form. In the case of the journaling support, although filesystems with journaling support are fully backwards compatible with non-journalled ext2 filesystems, the implementation required enough changes that the resulting filesystem has been named *ext3*.

The goal of this paper is to discuss how these features might be added to the filesystem in a coordinated fashion. Many of these new features are expected of modern filesystems; the challenge is to add them while maintaining ext2/3's advantages of a relatively small and simple code base, robustness in the face of I/O errors, and high levels of forwards and backwards compatibility.

2 Proposed enhancements to the ext2 filesystem format

We will discuss a number of extensions to the ext2/3 filesystem which will likely be implemented in the near future. For the most part, these extensions are independent of each other, and can be implemented in any order, although some extensions have synergistic effects. For example, two new features that will be described below, extent maps and persistent preallocation, are far more effective when used in combination with each other.

2.1 Directory indexing

Daniel Phillips has implemented a directory indexing scheme using a fixed-depth tree with hashed keys[3]. This replaces the linear directory search algorithm currently in use with traditional ext2 filesystems, and significantly improves performance for very large directories (thousands of files in a single directory).

The interior, or index, nodes in the tree are formatted to look like deleted directory entries, and the leaf nodes use the same format as existing ext2 directory blocks. As a result, read-only backwards compatibility is trivially achieved. Furthermore, starting in the Linux 2.2 kernel, whenever a directory is modified, the `EXT2_BTREE_FL` (since renamed `EXT2_INDEX_FL`) is cleared. This allows us to guarantee read/write compatibility with Linux 2.2 kernels, since the filesystem can detect that the internal indexing nodes are probably no longer consistent, and thus should be ignored until they can be reconstructed (via the `e2fsck` program).

Daniel Phillip's directory indexing code is currently available as a set of patches versus the 2.4 ext2 code base. As of this writing, the patches still need to be merged with the ext3 journaling code base. In addition, there are plans to select a better hash function that has better distribution characteristics for filenames commonly found in workloads such as mail queue directories. There are also plans to add hinting information in the interior nodes of the tree to indicate that a particular leaf node is nearly empty and that its contents could be merged with an adjacent leaf node.

2.2 On-line filesystem resizing

Andreas Dilger has implemented patches to the ext2 filesystem that support dynamically increasing the size

of the filesystem while the filesystem is on-line. Before logical volume managers (LVMs) became available for Linux, off-line resizing tools such as `resize2fs`, which required that the filesystem be unmounted and checked using `e2fsck` first, were sufficient for most users' needs. However, with the advent of LVM systems that allow block devices to be dynamically grown, it is much more important filesystems to be able to grow and take advantage of new storage space which has been made available by the LVM subsystem without needing to unmount the filesystem first. Indeed, administrators of enterprise-class systems take such capabilities for granted. (Dynamically shrinking mounted filesystems is a much more difficult task, and most filesystems do not offer this functionality. For ext2/3 filesystems, filesystems can be shrunk using the off-line resizing tool `resize2fs`.)

A disadvantage of the current ext2 resizing patches is that they require that the filesystem be prepared before the filesystem can be resized on-line. This preparation process, which must be done with the filesystem unmounted, finds the inodes using the blocks immediately following the block group descriptors, and relocates these blocks so they can be reserved for the resizing process. These blocks must be reserved since the current layout of the ext2 superblock and block group descriptors require an additional block group descriptor block for each 256MB, 2GB, or 16GB of disk space for filesystems with 1KB, 2KB, and 4KB block sizes, respectively. Although the requirement for an off-line preparation step is quite inconvenient, this scheme does have the advantage that the filesystem format remains unmodified, so it is fully compatible with kernels that do not support on-line resizing. Still, if the system administrator knows in advance how much a filesystem may need to be grown, reserving blocks for use by the block group descriptors may be a workable solution.

Requiring advance preparation of the filesystem can be obviated if we are willing to let the filesystem become incompatible with older kernels after it has been extended. Given that many 2.0 and 2.2 kernels do not support LVM devices (and so would be unable to read a filesystem stored on an LVM anyway), this may be acceptable. The change in the filesystem format replaces the current scheme where the superblock is followed by a variable-length set of block group descriptors. Instead, the superblock and a *single* block group descriptor block is placed at the beginning of the first, second, and last block groups in a *meta-block group*. A meta-block group is a collection of block groups which can be described by a single block group descriptor block. Since the size of the block group descriptor structure is 32 bytes, a

meta-block group contains 32 block groups for filesystems with a 1KB block size, and 128 block groups for filesystems with a 4KB blocksize. Filesystems can either be created using this new block group descriptor layout, or existing filesystems can be resized on-line, and a new field in the superblock will indicate the first block group using this new layout.

This new scheme is much more efficient, while retaining enough redundancy in case of hardware failures. Most importantly, it allows new block groups to be added to the filesystem without needing to change block group descriptors in the earlier parts of the disk. Hence, it should be very simple to write an ext2/3 filesystem extension using this design that provides on-line resizing capabilities.

2.3 An expanded inode

The size of the on-disk inode in the ext2/3 filesystem has been 128 bytes long during its entire lifetime. Although we have been very careful about packing as much information as possible into the inode, we are finally getting to the point where there simply is not enough room for all of the extensions that people would like to add to the ext2/3 filesystem.

Fortunately, just before the release of Linux 2.0, most of the work to allow for an expanded inode was added. As part of the changes to version 1 of the ext2 superblock, the size of the inode in the filesystem was added as a parameter in the superblock. The only restriction on the size of inode is that it must evenly divide the filesystem blocksize. Unfortunately, some safety-checking code which aborted the filesystem from being mounted if the inode size was not 128 bytes was never removed from the kernel. Hence, in order to support larger inodes, a small patch will have to be made to the 2.0, 2.2, and 2.4 kernels. Fortunately the change is simple enough that it should be relatively easy to get the change accepted into production kernels.

One of the most important features that requires additional space in the inode is the addition of sub-second resolution timestamps. This is needed because given today's very fast computers, storing file modification times with only second granularity is not sufficient for programs like *make*. (For example, if *make* can compile all of the object files for a library and create the library within a second, a subsequent *make* command will not be able to determine whether or not the library needs to be updated.)

Another limitation imposed by the current inode field sizes is the use of a 16 bits for *i_links_count*, which limits the number of subdirectories that can be created in a single directory. The actual limit of 32,000 is smaller than what is possible with an unsigned 16-bit field, but even if the kernel were changed to allow 65,535 subdirectories, this would be too small for some users or applications.

In addition, extra inode space can also enable support 64-bit block numbers. Currently, using 4KB blocks, the largest filesystem that ext2 can support is 16TB. Although this is larger than any commonly available individual disks, there certainly are RAID systems that export block devices which are larger than this size.

Yet another future application that may require additional storage inside the inode is support for *mandatory access control* [4] (MAC) or audit labels. The NSA SE (Security-Enhanced) Linux[5] implementation requires a single 32-bit field for both purposes; other schemes may require two separate 32-bit fields to encode MAC and audit label.

In order to maximize backwards compatibility, the inode will be expanded without changing the layout of the first 128 bytes. This allows for full backwards compatibility if the new features in use are themselves backwards compatible — for example, sub-second resolution timestamps.

Doubling the inode size from 128 bytes to 256 bytes gives us room for 32 additional 32-bit fields, which is a lot of extra flexibility for new features. However, the 32 new fields can be very quickly consumed by designers proposing filesystem extensions. For example, adding support for 64-bit block pointers will consume almost half of the new fields. Hence, allocation of these new inode fields will have to be very carefully done. New filesystem features which do not have general applicability, or which require a large amount of space, will likely *not* receive space in the inode; instead they will likely have to use Extend Attribute storage instead.

2.4 Extended attributes, access control lists, and tail merging

One of the more important new features found in modern filesystems is the ability to associate small amounts of custom metadata (commonly referred to as *Extended Attributes*) with files or directories. Some of the applications of Extended Attributes (EA) include Access Control Lists[6], MAC Security Labels[6], POSIX

Capabilities[6], DMAP/XDSM[7] (which is important for implementing Hierarchical Storage Management systems), and others.

Andreas Gruenbacher has implemented ext2 extensions which add support for Extended Attributes and Access Control Lists to ext2. These patches, sometimes referred to as the *Bestbits patches*, since they are available at web site <http://www.bestbits.at>, have been relatively widely deployed, although they have not yet been merged into the main-line ext2/3 code base.

The *Bestbits* implementation uses a full disk block to store each set of extended attributes data. If two or more inodes have an identical set of extended attributes, then they can share a single extended attribute block. This characteristic makes the *Bestbits* implementation extremely efficient for Access Control Lists (ACLs), since very often a large number of inodes will use the same ACL. For example, it is likely that inodes in a directory will share the same ACL. The *Bestbits* implementation allows inodes with the same ACL to share a common data structure on disk. This allows for a very efficient storage of ACLs, as well as providing an important performance boost, since caching shared ACLs is an effective way of speeding up access control checks, a common filesystem operation.

Unfortunately, the *Bestbits* design is not very well suited for generic Extended Attributes, since the EA block can only be shared if all of the extended attributes are identical. So if every inode has some inode-unique EA (for example, a digital signature), then each inode will need to have its own EA block, and the overhead for using EAs may be unacceptably high.

For this reason, it is likely that the mechanism for supporting ACLs may be different from the mechanisms used to support generic EAs. The performance requirements and storage efficiencies of ACL sharing justify seriously considering this option, even if it would be more aesthetically pleasing, and simpler, to use a single EA storage mechanism for both ACLs and generic EAs.

There may be a few other filesystem extensions which require very fast access by the kernel; for example, mandatory access control (MAC) and audit labels, which need to be referenced every time an inode is manipulated or accessed. In these cases, however, as mentioned in the previous section, the simplest solution is to reserve an extra field or two in the expanded ext2 inode for these applications.

One of more promising tactics for solving the EA stor-

age problem is to combine it with Daniel Phillips's proposal of adding *tail merging* to the ext2 filesystem. Tail merging is the practice of storing the data contained in partially filled blocks at the end of files (called tails) in a single shared block. This shared block could also be used as a location of storing Extended Attributes. In fact, tail-merging can be generalized so that a tail is simply a special Extended Attribute.

The topic of extended attributes is still a somewhat controversial area amongst the ext2 developers, for a number of reasons. First, there are many different ways in which EAs could be stored. Second, how EAs will be used is still somewhat unclear. Realistically, they are not used very often today, primarily because of portability concerns; EAs are not specified by any of the common Unix specifications: POSIX.1[8], SUS[9], etc., are not supported by file archiving tools such as *tar* and *zip*, and they cannot be exported over NFS (though the new NFSv4 standard[10] does include EA support.) Still, the best alternatives which seem to have been explored to date will probably keep the *Bestbits* approach exclusively for ACLs, and an approach where multiple inodes can utilize a single filesystem block to store tails and extended attributes.

However, progress is being made: the linux-2.5 kernel now includes a standard API for accessing ACLs, and the popular *Samba* file-serving application can already use that API, if it is present.

2.5 Extensible inode table

With the increase in size of the on-disk inode data structure, the overhead of the inode table naturally will be larger. This is compounded by the general practice of significantly over-provisioning the number of inodes in most Unix filesystems, since in general the number of inodes cannot be increased after the filesystem is created. While experienced system administrators may change the number of inodes when creating filesystems, the vast majority of filesystems generally use the defaults provided by *mke2fs*. If the filesystem can allocate new inodes dynamically, the overhead of the inode table can be reduced since there will no longer be a need to over-allocate inodes.

Expanding the inode table might seem to be a simple and straightforward operation, but there are a number of constraints that complicate things. We cannot simply increase the parameter indicating the number of inodes per block group, since that would require renumbering all of

the inodes in the filesystem, which in turn would require scanning and modifying all of the directory entries in the filesystem.

Also complicating matters is the fact that the inode number is currently used as part of the block and inode allocation algorithms. An inode's number, when divided by the filesystem's `inodes_per_block_group` parameter, results in the block group where the inode is stored. This is used as a hint when allocating blocks for that inode for better locality. Simply numbering new inodes just beyond the last used inode number will destroy this property. This presents problems especially if the filesystem may be dynamically resized, since growing the filesystem also grows the inode table, and the inode numbers used for the extensible inode table must not conflict with the inode numbers used when the filesystem is grown.

One potential solution would be to extend the inode number to be 64 bits, and then encode the block group information explicitly into the high bits of the inode number. This would necessarily involve an incompatible change to the directory entry format. However, if we expand the block pointers to 64 bits to support petabyte-sized filesystems, we ultimately may wish to support more than 2^{32} inodes in a filesystem anyway. Unfortunately, there are two major implementation problems with expanding the inode number which make pursuit of this approach unlikely. First, the size of the inode number in `struct stat` is 32 bits on 32-bit platforms; hence, user space programs which depend on different inodes having unique inode numbers may have this assumption violated. Secondly, the current ext2/3 implementation relies on internal kernel routines which assume a 32-bit inode number. In order to use a 64-bit inode number, these routines would have to be duplicated and modified to support 64-bit inode numbers.

Another potential solution to this problem is to utilize inode numbers starting from the end of the inode space (i.e., starting from $2^{32} - 1$ and working downwards) for dynamically-allocated inodes, and using an inode to allocate space for these extended inodes. For the purposes of the block allocation algorithm, the extended inode's block group affiliation can be stored in a field in the inode. However, the location of the extended inode in this scheme could no longer be determined by examining its inode number, so the location of the inode on disk would no longer be close to the data blocks of the inode. This would result in a performance penalty for using extended inodes (since the location of the inode and the location of its data blocks would no longer necessarily be close together), but hopefully the penalty would not be too great.

Some initial experiments which grouped the inode tables of *meta-block groups* together showed a very small performance penalty, although some additional benchmarking is necessary. (A simple experiment would be to modify the block allocation algorithms to deliberately allocate blocks in a different block group from the inode, and to measure the performance degradation this change would cause.)

2.6 Extent maps

The ext2 filesystem uses direct, indirect, double indirect, and triple indirection blocks to map file offsets to on-disk blocks, like most classical Unix filesystems. Unfortunately, the direct/indirect block scheme is inefficient for large files. This can be easily demonstrated by deleting a very large file, and noting how long that operation can take. Fortunately, ext2 block allocation algorithms tend to be very successful at avoiding fragmentation and in allocating contiguous data blocks for files. For most Linux filesystems in production use today, the percentage of non-contiguous files reported by `e2fsck` is generally less than 10%. This means that in general, over 90% of the files on an ext2 filesystem only require a single *extent map* to describe all of their data blocks. The extent map would be encoded in a structure like this:

```
struct ext2_extent {
    __u64    logical_block;
    __u64    physical_block;
    __u32    count;
};
```

Using such a structure, it becomes possible to efficiently encode the information, "Logical block 1024 (and following 3000 blocks) can be found starting at physical block 6536." The vast majority of files in a typical Linux system will only need a few extents to describe all of their logical to physical block mapping, and so most of the time, these extent maps could be stored in the inode's direct blocks.

However, extent maps do not work well in certain pathological cases, such as sparse files with random allocation patterns. There are two ways that we can deal with these sorts of cases. The traditional method is to store the extent maps in a B-tree or related data structure, indexed by the logical block number. If we pursue this option, it will not be necessary to use the full balancing requirements of B-trees; we can use similar design choices to those made by the directory indexing design

to significantly simplify a B-tree implementation: using a fixed depth tree, not rotating nodes during inserts, and not worrying about rebalancing the tree after operations (such as truncate) which remove objects from the tree.

There is however an even simpler way of implementing extents, which is to ignore the pathological case altogether. Today, very few files are sparse; even most DBM/DB implementations avoid using sparse files. In this simplification, files with one or two extents can store the extent information in the inode, using the fields that were previously reserved for the direct blocks in the inode. For files with more extents than that, the inode will contain a pointer to a single extent-map block. (The single extent-map block can look like a single leaf belonging to an extent-map tree, so this approach could be later extended to support a full extent-map tree if this proves necessary.) If the file contains more extent maps than can fit in the single extent-map block, then indirect, double-indirect, and triple-indirect blocks could be used to store the remainder of the block pointers.

This solution is appealing, since for the vast majority of files, a single extent map is more than sufficient, and there is no need to adding a lot of complexity for what is normally a very rare case. The one potential problem with this simplified solution is that for very large files (over 25 gigabytes on a filesystem using a 4KB blocksize), a single extent map may not be enough space if filesystem metadata located at the beginning of each block group is separating contiguous chunks of disk space. Furthermore, if the filesystem is badly fragmented, then the extent map may fill even more quickly, necessitating a fall back to the old direct/double indirect block allocation scheme. So if this simplification is adopted, preallocation becomes much more important to ensure that these large block allocations happen contiguously, not just for performance reasons, but to avoid overflowing the space in a single extent map block.

We can solve the first problem of metadata (inode tables, block and inode bitmaps) located at the beginning of each block group breaking up contiguous allocations by solved by moving all the metadata out of the way. We have tried implementing this scheme by moving the inode tables and allocation bitmaps to the beginning of a *meta-block group*. The performance penalty of moving the inode table slightly farther away from the data blocks related to it was negligible. Indeed, for some workloads, performance was actually slightly improved by grouping the metadata together. Making this change does not require a format change to the filesystem, but merely a change in the allocation algorithms used by the `mke2fs` program. However, the kernel does have some sanity-

checking code that needs to be removed so that the kernel would not reject the mount. A very simple patch to weaken the checks in `ext3_check_descriptors()` was written for the 2.4 kernel. Patches to disable this sanity check, as well as the inode size limitation, will be available for all commonly used Linux kernel branches at <http://e2fsprogs.sourceforge.net/ext2.html>.

2.7 Preallocation for contiguous files

For multimedia files, where performance is important, it is very useful to be able to ask the system to allocate the blocks in advance, preferably contiguously if possible. When the blocks are allocated, it is desirable if they do not need to be zeroed in advanced, since for a 4GB file (to hold a DVD image, for example), zeroing 4GB worth of pre-allocated blocks would take a long time.

Ext2 had support for a limited amount of preallocation (usually only a handful of blocks, and the preallocated blocks were released when the file was closed). Ext3 currently has no preallocation support at all; the feature was removed in order to make adding journaling support simpler. However, it is clear that in the future, we will need to add a more significant amount of preallocation support to the ext2/ext3 filesystem.

In order to notify the filesystem that space should be preallocated, there are two interfaces that could be used. The POSIX specification leaves explicitly undefined the behavior of `ftruncate()` when the argument passed to `ftruncate` is larger than the file's current size. However, the X/Open System Interface developed by the Austin Group[11] states if the size passed to `ftruncate()` is larger than the current file size, the file should be extended to the requested size. The ext2/ext3 can use `ftruncate` as a hint that space should be preallocated for the requested size.

In addition to multimedia files, there are also certain types of files whose growth characteristics require persistent preallocation beyond the close of the inode. Examples of such *slow-growth* files include log files and Unix mail files, which are appended to slowly, by different processes. For these types of files, the ext2 behavior of discarding preallocated blocks when the last file descriptor for an inode is closed is not sufficient. On the other hand, retaining preallocated blocks for all inodes is also not desirable, as it increases fragmentation and can tie up a large number of blocks that will never be used.

One proposal would be to allow certain directories and

files to be tagged with an attribute indicating that they are slow-growth files, and so the filesystem should keep pre-allocated blocks available for these files. Simply setting this flag on the `/var/log` and `/var/mail` directories (so that newly created files would also have this flag set, and be considered slow-growth files) would likely make a big difference. It may also be possible to heuristically determine that a file should be treated as a slow-growth file by noting how many times it has been closed, and then re-opened and had data appended to it. If this happens more than once or twice, we can assume that it would be profitable to treat the file as a slow-growth file. Files opened with the `O_APPEND` flag (which is rarely used for regular file I/O) could also be assumed to be have slow-growth characteristics.

The types of preallocation described above are all non-persistent preallocation schemes. That is, the pre-allocated blocks are released if the filesystem is unmounted or if the system is rebooted. It is also possible to implement persistent preallocations (which is required for `posix_fallocate`), where the blocks are reserved on disk, and but not necessarily pre-zeroed. To support this feature, a 64-bit field in the inode will have to be allocated out of the newly expanded ext2 inode. This field, called the *high watermark*, specifies the last address that has actually been written to by the user. Attempts to read from the inode past this point must cause a zero-filled page to be returned, in order to avoid a security problem of exposing previously written and deleted data. Of course, if the user seeks past the high watermark and writes a block, the kernel must at that point zero all of the blocks between the high watermark and the point where the write was attempted.

Persistent preallocation may not be very important, since few applications require guarantees about preallocated contiguous allocations (even in the face of an unexpected system shutdown). As a result, persistent preallocation will likely be a very low-priority item to implement. The benefits of allowing (non-persistent) preallocation in ext3 filesystems are far greater, since they address the allocation needs of both slow-growth log and mail spool files, as well as large multimedia files.

3 Compatibility issues

Whereas many of the new features described in this paper are fully backwards compatible, some of these proposed new features introduce various different types of incompatibility. For example, even though an older ker-

nel would be able to read a filesystem containing files with high watermark pointers to implement persistent preallocation, a kernel which did not know to check the high watermark pointer could return uninitialized data, which could be a security breach. Because of this security issue, the persistent preallocation feature must use a bit in the `incompat` compatibility bitmask in the superblock.

Moreover, there are some changes that simply require incompatible filesystem feature bits due to the fundamental changes in the filesystem format. A good example of such a feature is the extent map changes. Older kernels will not know how to interpret extent maps. In the past, when we have made incompatible changes, `e2fsprogs` has provided conversion utilities (usually as part of the `tune2fs` and `e2fsck` programs) to add and remove new features to filesystems.

Other changes, such as expanding the size of the on-disk inode structures, will require the use of technology already found in `resize2fs` to relocate data blocks belonging to inodes to other locations on disk to make room for growing system data areas.

Andreas Dilger has also suggested an interesting way of providing the largest amount of backwards compatibility as possible by adding compatibility flags on a per-inode basis. So if there are only a few files which are using persistent-preallocation or extent maps, the filesystem could be mounted without causing problems for the majority of the files which are not using those features.

Table 1 shows which of the proposed new ext2 features are backwards compatible and which are not. Each incompatible feature can be enabled or disabled on a per-filesystem (and perhaps per-inode basis); in addition, for many of these incompatible changes, it would be very simple to make backports of these features available to older kernels so that they would be able to use filesystems with some of these new features.

4 Implementation issues

Nearly all of the extensions described here can be implemented independently of the others. This allows for distributed development, which is consonant with the general Linux development model. The only real dependency that exists is that a number of the new features, such as subsecond timestamps, persistent preallocation, and 64-bit block numbers require an expanded inode.

Feature	Compatible?
Directory indexing	Y
On-line filesystem resizing	N
Expanded inode	Y
Subsecond timestamps	Y
<i>Bestbits</i> ACL	Y
Tail-merging	N
Extent maps	N
Preallocation	Y
Persistent preallocation	N

Table 1: Ext2/3 extensions compatibility chart

Hence, an early priority will be enhancing `resize2fs` so that it can double the size of the inode structure on disk. Another high priority task is to make available kernel patches for all commonly used kernel versions (at least for the 2.2 and 2.4 kernels) that remove the safety checks that prevent current kernels from mounting filesystems with expanded inodes. The sooner these patches are available, the sooner they can get adopted and installed on production systems. This will ease the transition and compatibility issues immensely.

4.1 Factorization of ext2/3 code

One of the reasons why we have separate code bases for ext2 and ext3 is that journaling adds a lot of complexity to a number of code paths, especially in the block allocation code. Factoring out these changes so that journaling and non-journaling variants of block allocation functions, inode modification routines, etc., could be selected via function pointers and an operations table data structure will clean up the ext2/3 implementation. This will allow us to have a single code base which can support filesystems both with and without journaling.

4.2 Source control issues

Now that as many as six developers are experimenting with various ext2/3 extensions, some kind of source control system is needed so that each developer could have their own source-controlled playground to develop their own changes, and also allow them to easily merge their changes with updates in the development tree. Up until now we have been using CVS. However, our experience with using CVS for maintaining ext3 kernel code has shown that CVS does not deal well with a large number of branches. Keeping track of a large number of

branches is very difficult under CVS; it does not have any native visualization tools, and merging changes between different branches is a manual process which is highly error-prone.

We have started using `bitkeeper` to maintain the `e2fsprogs` user space utilities, and this experiment has been very successful. In addition, the master 2.4 and 2.5 Linux kernels are being maintained using `bitkeeper`, as Linus Torvalds and many other kernel developers have found that it best fits the highly distributed nature of development of the Linux kernel. For these reasons, the authors are currently strongly exploring the possibility of using `bitkeeper` as the source control mechanism for the ext2/3 kernel code. The open-source subversion source control system may also be viable in the future: it promises good support for repeated merges between development branches, but it is still quite immature compared to `bitkeeper` and CVS.

5 Conclusions

In this paper, we have reviewed some of the extensions to the ext2/3 filesystem that are currently being planned. Some of these designs may change while the extensions are being implemented. Still, it is useful to work through design issues before attempting to put code to paper (or to emacs or vi buffers, as appropriate), since a number of these extensions interact with one another, and create dependencies amongst themselves.

In addition, there are number of other optimizations being planned for the Linux 2.5 kernel that are not strictly part of the ext2 filesystem, but which will significantly impact its performance. Examples of such planned optimizations in the VM layer include write-behind optimizations and support for the `O_DIRECT` open flag.

Other topics that we will likely explore in the future include allowing multiple filesystems to share a journal device, better allocation algorithms that take into account RAID configurations, and large (32KB or 64KB) block-sizes.

Finally, perhaps it would be appropriate to answer at this point a common question. Given that there are many new, modern filesystems such as XFS with advanced features, why are we working on adding new features to ext2/3? There are a number of answers to that question:

- Ext3 supports data journaling which can improve

performance for remote filesystems that require synchronous updates of data being written.

- Ext3 allows for a smooth upgrade facility for existing ext2 filesystems (of which there are many).
- The ext3 code base is fairly small and clean, and has an existing strong developer community that work at a variety of different companies. Hence, the future of ext2/3 is not tied to the success or failure of a single company, and a single company can not unduly influence the future of ext2/3.

6 Acknowledgments

There have many people who have worked on the ext2 and ext3 filesystems, and their contributions both in terms of code and design discussions have been invaluable. Those in particular who deserve special mention include Andrew Morton and Peter Braam (who both helped with the port of ext3 to 2.4), Daniel Phillips (who implemented the tail-merging and directory indexing patches), Andreas Dilger (who contributed numerous patches to ext3 and to e2fsprogs), and Al Viro (who has fixed up and significantly improved the truncate and directory page cache). All of these people also contributed extensively to discussions on the `ext2-devel@lists.sourceforge.net` mailing list, and helped to refine the design plans found in this paper. Thank you all very much indeed.

References

- [1] R. Card, T. Y. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proceedings of the 1994 Amsterdam Linux Conference*, 1994.
- [2] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, vol. 2, pp. 181–197, August 1984.
- [3] D. Phillips, "A Directory Index for Ext2," *Proceedings of the 2001 Annual Linux Showcase and Conference*, 2001.
- [4] Trusted Computer Security Evaluation Criteria, DOD 5200.28-STD. Department of Defense, 1985.
- [5] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", *Freenix Track: 2001 Usenix Annual Technical Conference*, 2001.
- [6] POSIX 1003.1e Draft Standard 17 (withdrawn), POSIX, 1997.
- [7] CAE Specification Systems Management: Data Storage Management (XDSM) API, The Open Group, 1997.
- [8] Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API), IEEE, 1996.
- [9] The Single Unix Specification, Version 2, The Open Group, 1997
- [10] S. Shepler and B. Callaghan and D. Robinson and R. Thurlow and C. Beame and M. Eisler and D. Noveck, "NFS Version 4 Protocol", RFC 3010, Internet Engineering Task Force, 2000.
- [11] The Open Group Base Specifications Issue 6: Systems Interface volume (XSI), The Open Group, 2001.
- [12] The Bitkeeper Distributed Source Management System, <http://www.bitkeeper.com>, 2002.

Recent Filesystem Optimisations in FreeBSD

Ian Dowse <iedowse@freebsd.org>

Corvil Networks.

David Malone <dwmalone@freebsd.org>

CNRI, Dublin Institute of Technology.

Abstract

In this paper we summarise four recent optimisations to the FFS implementation in FreeBSD: *soft updates*, *dirpref*, *vmiodir* and *dirhash*. We then give a detailed exposition of *dirhash*'s implementation. Finally we study these optimisations under a variety of benchmarks and look at their interactions. Under micro-benchmarks, combinations of these optimisations can offer improvements of over two orders of magnitude. Even real-world workloads see improvements by a factor of 2–10.

1 Introduction

Over the last few years a number of interesting filesystem optimisations have become available under FreeBSD. In this paper we have three goals. First, in Section 2, we give a summary of these optimisations. Then, in Section 3, we explain in detail the *dirhash* optimisation, as implemented by one of the authors. Finally we present the results of benchmarking various common tasks with combinations of these optimisations.

The optimisations which we will discuss are *soft updates*, *dirpref*, *vmiodir* and *dirhash*. All of these optimisations deal with filesystem metadata. *Soft updates* alleviate the need to make metadata changes synchronously; *dirpref* improves the layout of directory metadata; *vmiodir* improves the caching of directory metadata; and *dirhash* makes the searching of directories more efficient.

2 Summaries

For each optimisation we will describe the performance issue addressed by the optimisation, how it is addressed and the tradeoffs involved with the optimisation.

2.1 Soft Updates

Soft updates is one solution to the problem of keeping on-disk filesystem metadata recoverably consistent. Traditionally, this has been achieved by using synchronous writes to order metadata updates. However, the performance penalty of synchronous writes is high. Various schemes, such as journaling or the use of NVRAM, have been devised to avoid them [14].

Soft updates, proposed by Ganger and Patt [4], allows the reordering and coalescing of writes while maintaining consistency. Consequently, some operations which have traditionally been durable on system call return are no longer so. However, any applications requiring synchronous updates can still use `fsync(2)` to force specific changes to be fully committed to disk. The implementation of soft updates is relatively complicated, involving tracking of dependencies and the roll forward/back of transactions. Thus soft updates trades traditional write ordering, memory usage and code complexity for significantly faster metadata changes.

McKusick's production-quality implementation [8] has been found to be a huge win in real-world situations. A few issues with this implementation persist, such as failure to maintain NFS semantics. Long standing issues, such as disks appearing full because of outstanding transactions have recently been resolved.

In FreeBSD 5.0-current, soft updates has been combined with snapshots to remove the need for a full `fsck` at startup [7]. Instead, the filesystem can be safely preened while in use.

2.2 Dirpref

Dirpref is a modification to the directory layout code in FFS by Orlov [10]. It is named after the `ffs_dirpref` function, which expresses a preference for which inode should be used for a new directory.

The original policy implemented by this function was to select from among those cylinder groups with above the average number of free inodes, the one with the smallest number of directories. This results in directories being distributed widely throughout a filesystem.

The new dirpref policy favours placing directories close to their parent. This increases locality of reference, which reduces disk seek times. The improved caching decreases the typical number of input and output operations for directory traversal. The obvious risk associated with dirpref is that one section of the disk may become full of directories, leaving insufficient space for associated files. Heuristics are included within the new policy to make this unlikely. The parameters of the heuristic are exposed via `tunefs` so that they may be adjusted for specific situations.

2.3 Vmiodir

Vmiodir is an option that changes the way in which directories are cached by FreeBSD. To avoid waste, small directories were traditionally stored in memory allocated with `malloc`. Larger directories were cached by putting pages of kernel memory into the buffer cache. Malloced memory and kernel pages are relatively scarce resources so their use must be restricted.

Some unusual access patterns, such as the repeated traversal of a large directory tree, have a working set that consists almost exclusively of directories. Vmiodir enables direct use of the virtual memory system for directories, permitting maximal caching of such working sets, because cached pages from regular files can be flushed out to make way for directories.

Here the trade-off is that while some memory is wasted because 512-byte directories take up a full physical page, the VM-backed memory is better managed and more of it is usually available. Once a modest amount of memory is present in the system, this results in better caching of directories. Initially added to FreeBSD in June 1999 by Dillon, vmiodir was disabled by default until real-world use confirmed that it was usually at least as good as the more frugal scheme.

2.4 Dirhash

A well-known performance glitch in FFS is encountered when directories containing a very large number of en-

tries are used. FFS uses a linear search through the directory to locate entries by name and to find free space for new entries. The time consumed by full-directory operations, such as populating a directory with files, can become quadratic in the size of the directory.

The design of filesystems since FFS has often taken account of the possibility of large directories, by using B-Trees or similar structures for the on-disk directory images (e.g., XFS [16], JFS [1] and ReiserFS [13]).

Dirhash retrofits a directory indexing system to FFS. To avoid repeated linear searches of large directories, dirhash builds a hash table of directory entries on the fly. This can save significant amounts of CPU time for subsequent lookups. In contrast to filesystems originally designed with large directories in mind, these indices are not saved on disk and so the system is backwards compatible.

The cost of dirhash is the effort of building the hash table on the first access to the directory and the space required to hold that table in memory. If the directory is accessed frequently then this cost will be reclaimed by the saving on the subsequent lookups.

A system for indexing directories in Linux's Ext2 filesystem has also recently been developed by Phillips [11]. This system saves the index on disk in a way that provides a good degree of backwards compatibility.

3 Inside Dirhash

Dirhash was implemented by Ian Dowse in 2001 in response to discussions on the `freebsd-hackers` mailing list. Dirhash builds a summary of the directory on its first access and then maintains that summary as the directory is changed. Directory lookup is optimised using a hash table and the creation of new directory entries is optimised by maintaining an array of free-space summary information.

Directories are hashed only if they are over a certain size. By default this size is 2.5KB, a figure which should avoid wasting memory on smaller directories where performance is not a problem. Hashing a directory consumes an amount of memory proportional to its size. So, another important parameter for dirhash is the amount of memory it is allowed to use. Currently it works with a fixed-size pool of memory (by default 2MB).

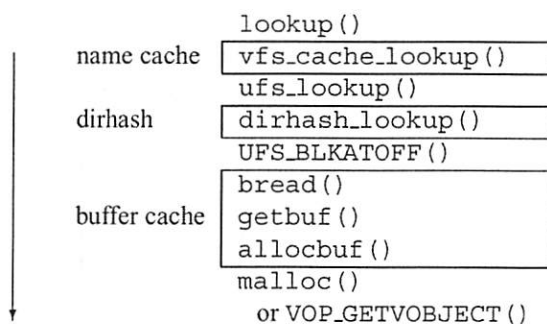


Figure 1: Directory Caching in FreeBSD. The major caches affecting directory lookups are shown on the left and the corresponding call stack is shown on the right.

Both these parameters of dirhash are exposed by sysctl, in addition to another option that enables extra sanity checking of dirhash's summary information.

3.1 Existing Directory Caching in FreeBSD

The translation of filenames to vnodes is illustrated in Figure 1. An important part of this process is the name cache, implemented in `vfs_cache.c`. Here a global hash table is used to cache the translation of <directory vnode, path component> pairs to vnodes.

A cache entry is added when a component is first looked up. Cache entries are made for both successful and failed lookups. Subsequent lookups then avoid filesystem-level directory searches, unless access to the underlying directory is required.

Name cache entries persist until the associated vnodes are reclaimed. Cache entries for failed lookups are limited to a fixed proportion of the total cache size (by default 1/16). Some operations result in the entries relating to a particular vnode or a particular filesystem being explicitly purged.

The name cache can help in the case of repeated accesses to existing files (e.g., Web serving) or accesses to the same non-existent file (e.g., Apache looking for `.htaccess` files). The name cache cannot help when the filename has not been previously looked up (e.g., random file accesses) or when the underlying directory must be changed (e.g., renaming and removing files).

3.2 Dirhash Based Name Lookups

Like the name cache, dirhash also maintains a hash table for mapping component names to offsets within the directory. This hash table differs from the global name cache in several ways:

- The tables are per directory.
- The table is populated on creation by doing a pass over the directory, rather than populated as lookups occur.
- The table is a complete cache of all the directory's entries and can always give a definitive yes or no answer whereas the name cache contains only the positive and negative entries added by lookups.
- Dirhash stores just the offset of directory entries within its table. This is sufficient to read the appropriate directory block from the buffer cache, which contains all the information needed to complete the lookup. In contrast, the VFS name cache is a filesystem-independent heavyweight cache storing the component name and references to both directory and target vnodes within its hash table.

Since dirhash only needs to store an offset, it uses an open-storage hash table rather than using linked lists (as the name cache does). Use of linked lists would increase the table size by a factor of two or three. Hash collisions within the table are resolved using linear-probing. This has the advantage of being simple and, unlike some other open storage schemes, some deleted entries can be reclaimed without the need to rebuild the table. Specifically, deleted entries may be marked as empty when they lie at the end of a chain.

Due to the initial population of the dirhash table, filling the name cache with a working set of m files from a directory of size n should cost $n + m$ rather than nm for random accesses without dirhash. If a working set of files is established in the name cache, then the dirhash hash table will no longer be accessed. The case of sequential directory access was traditionally optimised in the FFS code and would have given a cost of m . A sequential lookup optimisation is also present in dirhash; after each successful lookup, the offset of the following entry is stored. If the next lookup matches this stored value then the sequential optimisation is enabled. While the optimisation is enabled, dirhash first scans through the hash chain looking for the offset stored by the previous lookup. If the offset is found, then dirhash consults that offset before any other.

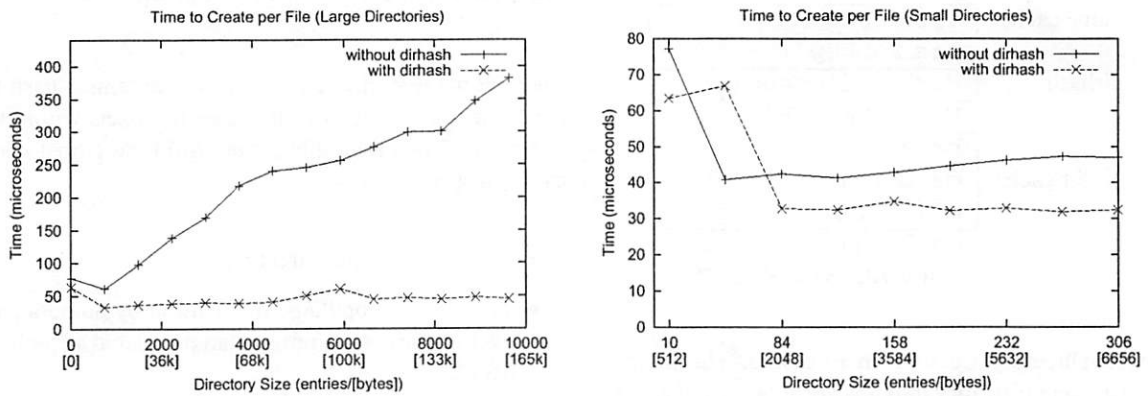


Figure 2: File Creation: Cost per Creation

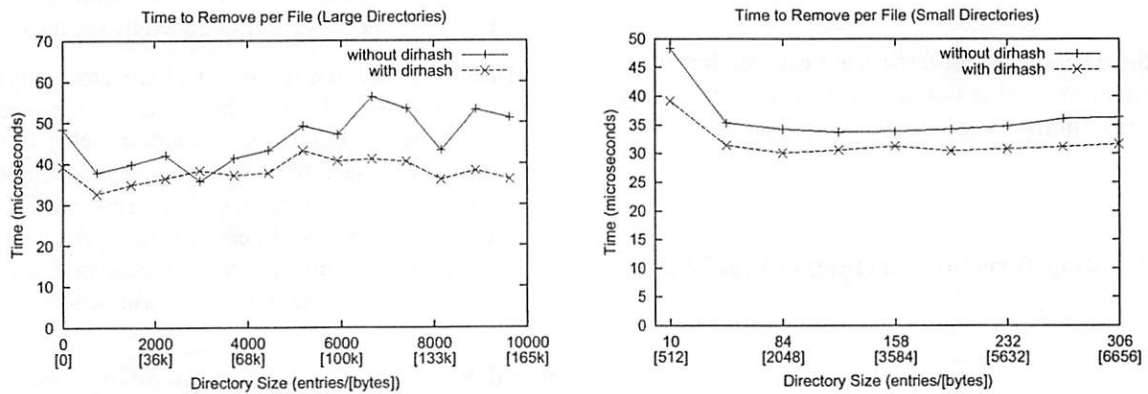


Figure 3: File Removal: Cost per Removal

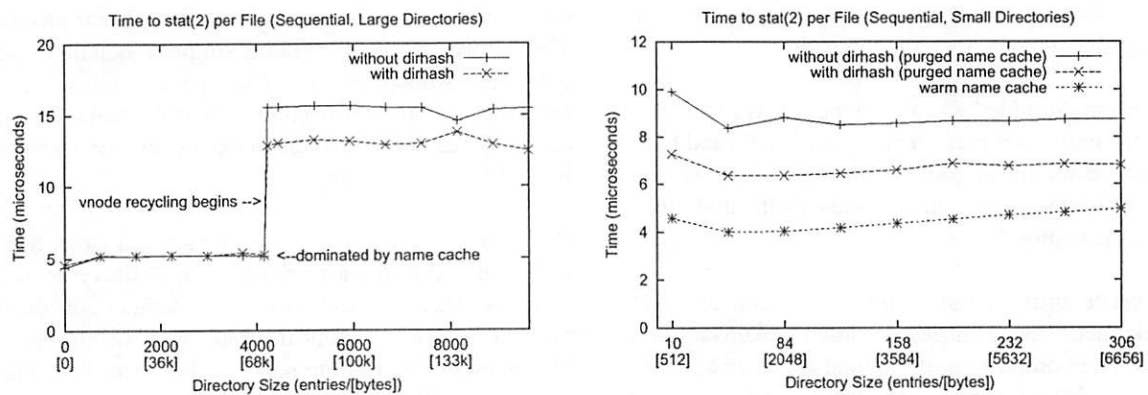


Figure 4: Sequential stat(2) of Files: Cost per stat(2)

A comparison between cost per operation of the traditional linear scheme and dirhash for file creation and removal is shown in Figures 2 and 3 respectively. The costs for large directories are shown on the left and those for small directories are on the right. All these tests were performed with our benchmarking hardware described in Section 4.2. For creation with the linear scheme, we

can see the cost of creations increasing as the directory size increases. For dirhash the cost remains constant at about $50\mu s$. Files were removed in sequential order, so we expect the cost to be roughly constant for both schemes. This is approximately what we see (dirhash around $35\mu s$, linear around $45\mu s$), though the results are quite noisy (perhaps due to delayed operations).

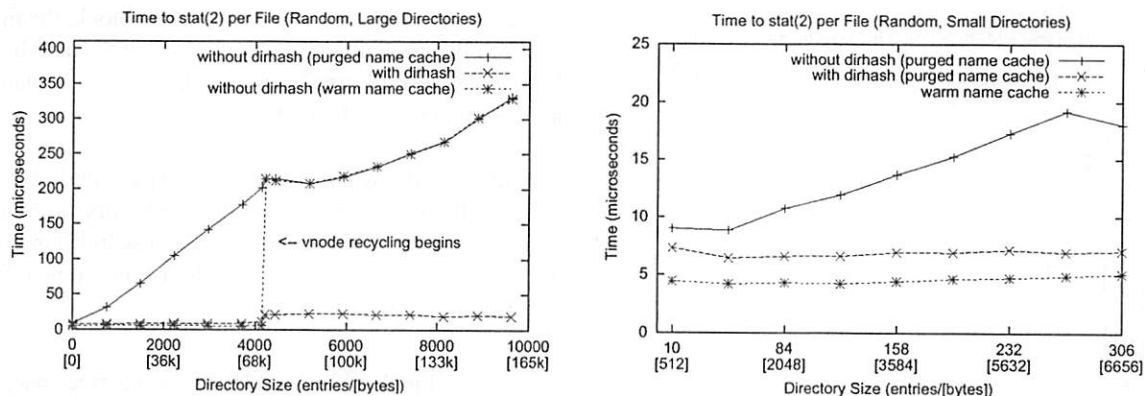


Figure 5: Random stat(2) of Files: Cost per stat(2)

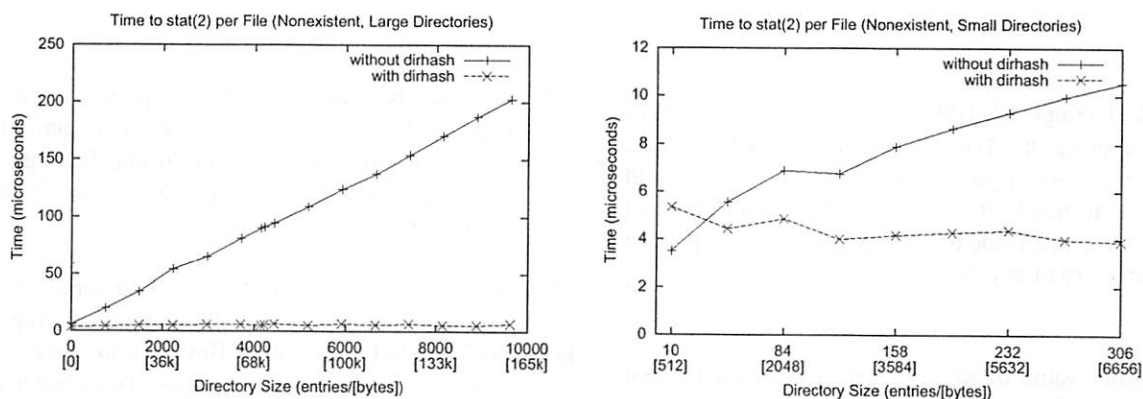


Figure 6: stat(2) of Nonexistent Files: Cost per stat(2)

The cost of stat operations, both with and without dirhash, are shown in Figure 4 for sequentially ordered files, in Figure 5 for randomly ordered files and Figure 6 for nonexistent files. Where the name cache has a significant impact on the results, we show figures for both a purged cache and a warm cache. Sequential operation here shows a cost of about $5\mu s$ if the result is in the name cache, $7\mu s$ for dirhash and $9\mu s$ for the linear search. For random ordering, results are similar for the name cache and dirhash. Here the linear search shows the expected linear growth rate. Both these graphs show an interesting jump at about 4000 files, which is explained below. Operating on nonexistent files incurs similar costs to operating on files in a random order. As no lookups are repeated, the name cache has no effect.

As mentioned, only directories over 2.5KB are usually indexed by dirhash. For the purposes of our tests, directories of any size were considered by dirhash, allowing us to assess this cut-off point. It seems to have been a relatively good choice, as dirhash look cheaper in all cases at 2KB and above. However, in practice, directories will be more sparsely populated than in these tests as

file creation and removal will be interleaved, so 2.5KB or slightly higher seems like a reasonable cut-off point.

The elbow in Figure 2 and the jump in Figures 4 and 5 at around 4000 vnodes is caused by an interaction between the test used, the way vnodes are recycled and the name cache:

1. For this test, stat is called on the files. Files are not held open nor read, meaning that no reference is held on the vnode in the kernel.
2. Consequently the vnodes end up on a ready-to-reuse list, but otherwise remain valid. Once there are `kern.minvnodes` vnodes allocated in the system, vnodes on the ready-to-reuse list are used in preference to allocating new vnodes. At this stage the vnode is recycled and its old contents becomes invalid.
3. The invalidation of vnodes purges the name cache of entries referring to those vnodes.

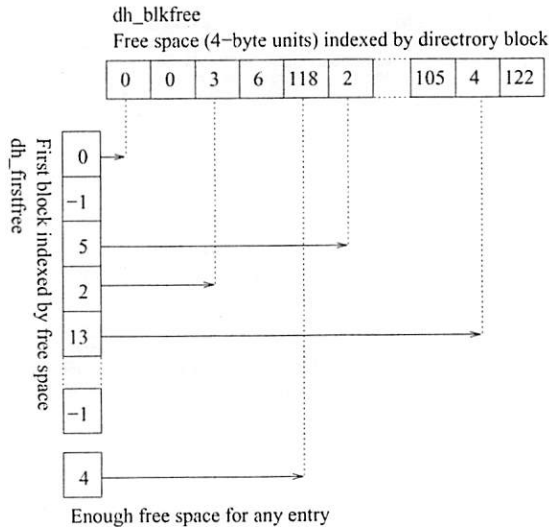


Figure 7: Example `dh_blkfree` and `dh_firstfree`. The free space in block is `dh_blkfree[block]`. `dh_firstfree[space]` gives the first block with exactly that much free space. The final entry of `dh_firstfree` is the first block with enough space to create an entry of any size.

The default value of `kern.minvnodes` on the test system is 4458. Altering this value causes the jump in the graph to move accordingly. In the usual situation where files are open and data is in the buffer cache, vnode reuse begins much later.

3.3 Locating Free Space with Dirhash

Dirhash also maintains two arrays of free-space summary information (see Figure 7 for an example). The first array, `dh_blkfree`, maps directory block numbers to the amount of free space in that block. As directory blocks are always 512 bytes and the length of entries is always a multiple of 4 bytes, only a single byte per block is required.

The second array, `dh_firstfree`, maps an amount of free space to the first block with exactly that much free space, or to `-1` if there is no such block. The largest legal FFS directory entry has a size of 264 bytes. This limits the useful length of the `dh_firstfree` array because there is never a need to locate a larger slot. Hence, the array is truncated at this size and the last element refers to a block with at least enough space to create the largest directory entry.

As entries are allocated and freed within a block, the first array can be recalculated without rescanning the whole block. The second array can then be quickly updated based on changes to the first.

Without dirhash, when a new entry is created, a full sweep of the directory ensures that the entry does not already exist. As this sweep proceeds, a search for suitable free space is performed. In order of preference, the new entry is created in:

- the first block with enough contiguous free space;
- the first block with enough free space but requires compaction;
- a new block.

When dirhash is used, the full sweep is no longer necessary to determine if the entry exists and the `dh_firstfree` array is consulted to find free space. The new entry is created in the first block listed in this array that contains enough free space.

These two policies are quite different, the former preferring to avoid compaction and the latter preferring to keep blocks as full as possible. Both aim to place new entries toward the start of the directory. This change of policy does not seem to have had any adverse effects.

3.4 Memory Use, Rehashing and Hash Disposal

Dirhash uses memory for the dirhash structure, the hash table and the free-space summary information. The dirhash structure uses 3 pointers, 9 ints, 1 `offset` and an array of 66 ints for `dh_firstfree`, about 320 bytes in total.

The hash table is initially sized at 150% of the maximum number of entries that the directory could contain (about 1 byte for every 2 bytes of the directory's size). The table used is actually two-level, so an additional pointer must be stored for every 256 entries in the table, or one pointer for every 2KB on the disk. The `dh_blkfree` array is sized at 150% of the blocks used on the disk, about 1 byte for every 340 on disk. This gives a total memory consumption of $266 \text{ bytes} + 0.505 \text{ dirsize}$ on a machine with 32-bit pointers or $278 \text{ bytes} + 0.507 \text{ dirsize}$ on a 64-bit platform.

The dirhash information is just a cache, so it is safe to discard it at any time and it will be rebuilt, if necessary,

on the next access. In addition to being released when the directory's vnode is freed, the hash is freed in the following circumstances:

- Adding an entry would make the table more than 75% full. Due to the linear probing algorithm, some hash table slots for deleted entries may not be reclaimed and marked empty. These are also counted as "used" slots.
- The directory grows too large for the number of entries in the `dh_blkfree` array.
- The directory shrinks so that the `dh_blkfree` array is less than one eighth used. This corresponds to the directory being truncated to about 20% of its size at the time the hash was built.
- The hash is marked for recycling.

Hashes may be marked for recycling when dirhash wants to hash another directory, but has reached its upper memory limit. If dirhash decided to recycle other hashes, then the hash table and the `dh_blkfree` array are released, but the dirhash structure remains allocated until the next access to the corresponding directory. This simplifies the interaction between the locking of the vnode and the locking of the dirhash structure.

To avoid thrashing when the working set is larger than the amount of memory available to dirhash, a score system is used which achieves a hybrid of least-recently-used (LRU) and least-frequently-used (LFU). This is intended to limit the rate of hash builds when the working set is large.

3.5 Implementation Details and Issues

The implementation of dirhash is relatively straightforward, requiring about 100 lines of header and 1000 lines of C. The actual integration of the dirhash code was unobtrusive and required only small additions to the surrounding FFS code at points where lookups occurred and where directories were modified. This means it should be easy for other systems using FFS to import and use the code.

The hash used is the FNV hash [9], which is already used within the FreeBSD kernel by the VFS cache and in the NFS code. The hash table is maintained as a two-level structure to avoid the kernel memory fragmentation that could occur if large contiguous regions were

allocated. The first level is an array of pointers allocated with `malloc`. These point to fixed-size blocks of 256 offsets, which are currently allocated with the zone allocator.

Some of the issues that emerged since the initial integration of dirhash include:

Expanded size of in-core inode: In FreeBSD-4, the addition of the dirhash pointer to the in-core inode structure resulted in the effective size of an in-core inode going from 256 bytes to 512 bytes. This exacerbated an existing problem of machines running out of memory for inodes. Some of the fields in the inode structure were rearranged to bring the size back down to 256 bytes.

Unexpected directory states: Dirhash's sanity checking code was too strict in certain unusual cases. In particular, `fsck` creates mid-block directory entries with inode number zero; under normal filesystem operation this will not occur. Also, dirhash has to be careful not to operate on deleted directories, as `ufs_lookup` can be called on a directory after it has been removed.

Bug in sequential optimisation: The sequential optimisation in dirhash was not functioning correctly until mid-November 2001, due to a typo which effectively disabled it. Once correctly enabled, sequential lookups seem a little faster (5%) rather than a little slower (1–2%) when compared to sequential non-dirhash lookups. This is probably a combination of two factors. First the non-dirhash code remembers the last block in which there was a successful lookup, but dirhash remembers the exact offset of the entry. Thus the non-dirhash code must search from the beginning of the block. Second, if the entry was at the end of a block, then the traditional optimisation may result in two blocks being fetched from the buffer cache, rather than one.

dh_firstfree bug: The last entry of the `dh_firstfree` array was not always updated correctly, unless there was a block with exactly enough space for the largest directory entry. This resulted in unnecessarily sparse directories.

More aggressive cleaning of deleted entries: If the deletion of a hash entry resulted in a chain ending with an empty slot, then dirhash can mark all the slots at the end of the chain as empty. Originally it was only marking the slots after the deleted entry as empty.

Improved hash function: Originally, the final stage of the hash function was to xor the last byte of the filename into the hash. Consequently, filenames differing only in the last byte end up closer together in the hash table. To optimise this common case, the address of the dirhash structure is hashed after the filename. This results in slightly shorter hash chains and also provides some protection against hash collision attacks.

The last three issues were uncovered during the writing of this paper (the version of dirhash used for the benchmarks predates the resolution of these issues).

On investigation, several dirhash bug reports have turned out to be faulty hardware. Since dirhash is a redundant store of information it may prove to be an unwitting detector of memory or disk corruption.

It may be possible to improve dirhash by storing hashes in the buffer cache and allowing the VM system to regulate the amount of memory which can be used. Alternatively, the slab allocator introduced for FreeBSD-5 may provide a way for dirhash to receive feedback about memory demands and adjust its usage accordingly. These options are currently under investigation.

Traditionally, directories were only considered for truncation after a create operation, because the full sweep required for creations was a convenient opportunity to determine the offset of the last useful block. Using dirhash it would be possible to truncate directories when delete operations take place, instead of waiting for the next create operation. This has not yet been implemented.

3.6 Comparisons with Other Schemes

The most common directory indexing technique deployed today is probably the use of on-disk tree structures. This technique is used in XFS [16] and JFS [1] to support large directories; it is used to store all data in ReiserFS [13]. In these schemes, the index is maintained on-disk and so they avoid the cost of building the index on first access. The primary downsides to these schemes are code complexity and the need to accommodate the trees within the on-disk format.

Phillips's HTree system [11] for Ext2/3 is a variant of these schemes designed to avoid both downsides. HTree uses a fixed-depth tree and hashes keys before use to achieve a more even spread of key values.

The on-disk format remains compatible with older versions of Ext2 by hiding the tree within directory blocks that are marked as containing no entries. The disk format is constructed cleverly so that likely changes made by an old kernel will be quickly detected and consistency checking is used to catch other corruption.

Again, HTree's index is persistent, avoiding the cost of dirhash's index building. However, the placing of data within directory blocks marked as empty is something which should not be done lightly as it may reduce robustness against directory corruption and cause problems with legacy filesystem tools.

Persistent indexes have the disadvantage that once you commit to one, you cannot change its format. Dirhash does not have this restriction, which has allowed the changing of the hash function without introducing any incompatibility. HTree allows for the use of different hashes by including a version number. The kernel can fall back to a linear search if it does not support the hash version.

Another obstacle to the implementation of a tree-based indexing scheme under FFS is the issue of splitting a node of the tree while preserving consistency of the filesystem. When a node becomes too full to fit into a single block it is split across several blocks. Once these blocks are written to the disk, they must be atomically swapped with the original block. As a consequence, the original block may not be reused as one of the split blocks. This could introduce complex interactions with traditional write ordering and soft updates.

A directory hashing scheme that has been used by NetApp Filers is described by Rakitzis and Watson, [12]. Here the directory is divided into a number of fixed length chunks (2048 entries) and separate fixed-size hash tables are used for each chunk (4096 slots). Each slot is a single byte indicating which 1/256th of the chunk the entry resides in. This significantly reduces the number of comparisons needed to do a lookup.

Of the schemes considered here, this scheme is the most similar to dirhash, both in design requirements (faster lookups without changing on-disk format) and implementation (it just uses open storage hash tables to implement a complete name cache). In the scheme described by Rakitzis and Watson the use of fixed-size hash tables, which are at most 50% full, reduces the number of comparisons by a factor of 256 for large directories. This is a fixed speed-up by a factor $N = 256$. As pointed out by Knuth [6], one attraction of hash tables is that as the number of entries goes to infinity the search time for

hash table stays bounded. This is not the case for tree based schemes. In principal dirhash is capable of these larger speed-ups, but the decision of Rakitzis and Watson to work with fixed-size hash tables avoids having to rebuild the entire hash in one operation.

4 Testimonial and Benchmark Results

We now present testimonial and benchmark results for these optimisations, demonstrating the sort of performance improvements that can be gained. We will also look at the interaction between the optimisations.

First, our testimonials: extracting the X11 distribution and maintaining a large MH mailbox. These are common real-world tasks, but since these were the sort of operations that the authors had in mind when designing dirpref and dirhash, we should expect clear benefits.

We follow with the results of some well-known benchmarks: Bonnie++ [2], an Andrew-like benchmark, NetApp's Postmark [5], and a variant of NetApp's Netnews benchmark [15]. The final benchmark is the running of a real-world workload, that of building the FreeBSD source tree.

4.1 The Benchmarks

The tar benchmark consisted of extracting the X410src-1.tgz file from the XFree [17] distribution, running `find -print` on the resulting directory tree and then removing the directory tree.

The folder benchmark is based on an MH inbox which has been in continuous use for the last 12 years. It involves creating 33164 files with numeric names in the order in which they were found to exist in this mailbox. Then the MH command `folder -pack` is run, which renames the files with names 1-33164, keeping the original numerical order. Finally the directory is removed.

Bonnie++ [2] is an extension of the Bonnie benchmark. The original Bonnie benchmark reported on file read, write and seek performance. Bonnie++ extends this by reporting on file creation, lookup and deletion. It benchmarks both sequential and random access patterns. We used version 1.01-d of Bonnie++.

The Andrew filesystem benchmark involves 5 phases:

creating directories, copying a source tree into these directories, recursive `stat(2)` of the tree (`find -exec ls -l` and `du -s`), reading every file (using `grep` and `wc`), and finally compilation of the source tree. Unfortunately, the original Andrew benchmark is too small to give useful results on most modern systems, and so scaled up versions are often used. We used a version which operates on 100 copies of the tree. We also time the removal of the tree at the end of the run.

Postmark is a benchmark designed to simulate typical activity on a large electronic mail server. Initially a pool of text files is created, then a large number of *transactions* are performed and finally the remaining files are removed. A transaction can either be a file create, delete, read or append operation. The initial number of files and the number of transactions are configurable. We used version 1.13 of Postmark.

The Netnews benchmark is the simplified version of Karl Swartz's [15] benchmark used by Seltzer et al. [14]. This benchmark initially builds a tree resembling inn's [3] traditional method of article storage (tradsPOOL). It then performs a large number of operations on this tree including linking, writing and removing files, replaying the actions of a news server.

As an example of the workload caused by software development, the FreeBSD build procedure was also used. Here, a copy of the FreeBSD source tree was checked out from a local CVS repository, it was then built using `make buildworld`, then the kernel source code searched with `grep` and finally the object tree was removed.

4.2 Benchmarking Method

The tests were conducted on a modern desktop system (Pentium 4 1.6GHz processor, 256MB ram, 20GB IDE hard disk), installed with FreeBSD 4.5-PRERELEASE. The only source modification was an additional `sysctl` to allow the choice between the old and new dirpref code.

Each benchmark was run with all 16 combinations of soft updates on/off, vmiodir on/off, dirpref new/old and dirhash maxmem set to 2MB or 0MB (i.e., disabling dirhash). Between each run a sequence of `sync` and `sleep` commands were executed to flush any pending writes. As the directory in which the benchmarking takes place is removed between runs, no useful data could be cached.

This procedure was repeated several times and the mean and standard deviation of the results recorded. Where rates were averaged, they were converted into time-per-work before averaging and then converted back to rates. A linear model was also fitted to the data to give an indication of the interaction between optimisations.

In some cases minor modifications to the benchmarks were needed to get useful results. Bonnie++ does not usually present the result of any test taking less than a second to complete. This restriction was removed as it uses `gettimeofday()` whose resolution is much finer than one second. Recording the mean and standard deviation over several runs should help confirm the validity of the result.

Postmark uses `time()`, which has a resolution of 1 second. This proved not to be of high enough resolution in several cases. Modifications were made to the Postmark timing system to use `gettimeofday()` instead.

4.3 Analysis of Results

The analysis of the results was something of a challenge. For each individual test there were 4 input parameters and one output parameter, making the data 5 dimensional. The benchmarks provided 70 individual test results.

First analysis attempts used one table of results for each test. The table's rows were ranked by performance. A column was assigned to each optimisation and presence of an optimisation was indicated by a bullet in the appropriate column. An example of such a table is shown in Table 1.

To determine the most important effects, look for large vertical blocks of bullets. In our example, soft updates is the most important optimisation because it is always enabled in the top half of the table. Now, to identify secondary effects look at the top of the upper and lower halves of the table, again searching for vertical blocks of bullets. Since `dirpref` is at the top of both the upper and lower halves, we declare it to be the second most important factor. Repeating this process again we find `vmiodir` is the third most important factor. `Dirhash`, the remaining factor, does not seem to have an important influence in this test.

The table also shows the time as a percentage, where no optimisations is taken to be 100%. The standard deviation and the number of runs used to produce the mean

SU	DP	VM	DH	Mean	Time%	σ/\sqrt{n}	n
•	•	•	•	2032.85	46.78	0.31	4
•	•	•		2058.92	47.38	0.32	4
•	•			2067.24	47.57	0.40	4
•	•		•	2118.72	48.76	0.03	4
•		•	•	2386.64	54.92	0.06	4
•		•		2389.99	55.00	0.05	4
•			•	2479.80	57.07	0.08	4
•				2489.56	57.29	0.04	4
	•	•	•	3681.77	84.73	0.19	4
	•	•		3697.03	85.08	0.44	4
	•			3724.66	85.72	0.17	4
	•		•	3749.88	86.30	0.24	4
		•	•	4260.20	98.04	0.06	4
		•		4262.49	98.09	0.02	4
				4345.38	100.00	0.05	4
			•	4348.22	100.07	0.08	4

Table 1: Rank Table of Results for Buildworld

result are also shown.

An attempt to analyse these results statistically was made by fitting to the data the following linear model, containing all possible interactions:

$$t = C_{\text{base}} + C_{\text{SU}}\delta_{\text{SU}} + C_{\text{DP}}\delta_{\text{DP}} + C_{\text{VM}}\delta_{\text{VM}} + C_{\text{DH}}\delta_{\text{DH}} + \text{Coefficients for pairs, triples, ...}$$

Here C_{opts} is a coefficient showing the effect of combining those optimisations and δ_{opt} is 1 if the optimisation is enabled and 0 otherwise. Finding the coefficients is a standard feature of many statistics packages.

Table 2 shows the results of fitting such a model to using the same input data used to build Table 1. Only coefficients that have a statistical significance of more than 1/1000 are shown. The coefficients suggests a 42.71% saving for soft updates, a 14.28% saving for `dirpref` and a 1.91% saving for `vmiodir`. The presence of a positive coefficient for the combination of `dirpref` and soft updates means that their improvements do not add directly, but overlap by 4.57 percentage points.

If a coefficient for a combination is negative, it means that the combined optimisation is more than the sum of its parts. An example is shown in Table 3, where the benefits of `dirhash` are dependent on the presence of soft updates (see Section 4.4 for more details).

In some cases the results of fitting the linear model was not satisfactory as many non-zero coefficients might be produced to explain a feature more easily understood by looking at a rank table, such as the one in Table 1, for that test.

Attempts to graphically show the data were also made,

Factor	Δ %Time	σ
base	100.00	0.24
DP	-14.28	0.34
VM	-1.91	0.34
SU	-42.71	0.34
DP : SU	4.57	0.48

Table 2: Linear Model Coefficients: Buildworld

Factor	Δ %Time	σ
base	100.00	0.01
SU	-98.67	0.01
DH : SU	-0.96	0.02

Table 3: Linear Model Coefficients: Removing MH mailbox

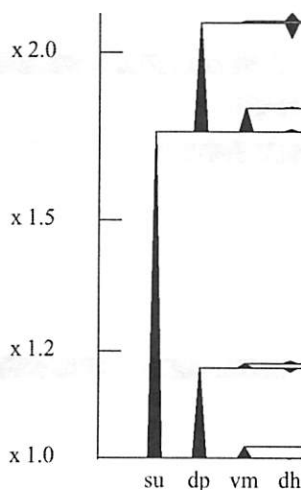


Figure 8: Improvement by Optimisation for Buildworld

but here the 5 dimensional nature of the data is a real obstacle. Displaying the transition between various combinations of optimisations was reckoned to be most promising. A decision was also made to display the data on a log scale. This would highlight effects such as those observed while removing an MH mailbox (Table 3), where soft updates reduces the time to 1.3% and then dirhash further reduces the time to 0.3%, which is a *relative* improvement of around 70%.

Figure 8 again shows the results of the build phase of the Buildworld benchmark, this time using a graphical representation where optimisations are sorted left to right by size of effect. The left-most triangle represents the benefit of enabling soft updates. The next column (two triangles) represents the effects of dirpref, starting with soft updates on and off. The third column (four triangles) represents vmidir and the right-most triangles rep-

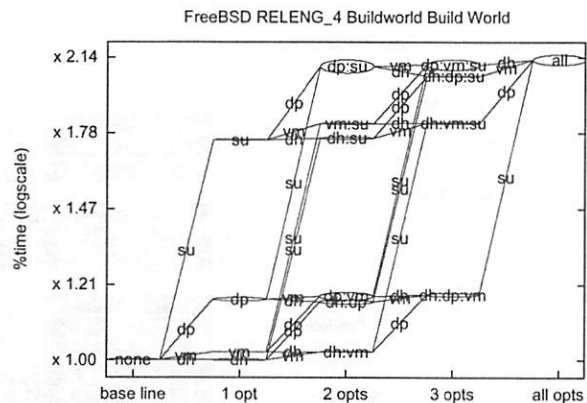


Figure 9: Improvement Lattice for Buildworld

resent dirhash. Most of these triangles point up, representing improvements, but some point down, representing detrimental effects.

These graphs make it relatively easy to identify important effects and relative effects as optimisations combine. They have two minor weaknesses however. First, not all transitions are shown in these graphs. Second, the variance of results is not shown.

In an effort to produce a representation that shows both of these factors, a graph of nodes was drawn, representing the various combinations of optimisations. An edge was drawn from one node to another to represent the enabling of individual optimisations. The x coordinate of the nodes indicates the number of enabled optimisations and the y coordinate the performance. The height of nodes shows the variance of the measurement.

Figure 9 shows the graph corresponding to the build phase of the Buildworld benchmark. There are many rising edges marked with soft updates and dirpref, indicating that these are significant optimisations. Although these graphs contain all the relevant information, they are often cluttered, making them hard to read.

4.4 Summary of Results

Figure 10 presents a summary of the benchmark results (the Bonnie++ file I/O and CPU results have been omitted for reasons of space). Studying the results indicates that soft updates is the most significant factor in almost all the benchmarks. The only exceptions to this are benchmarks that do not involve writing.

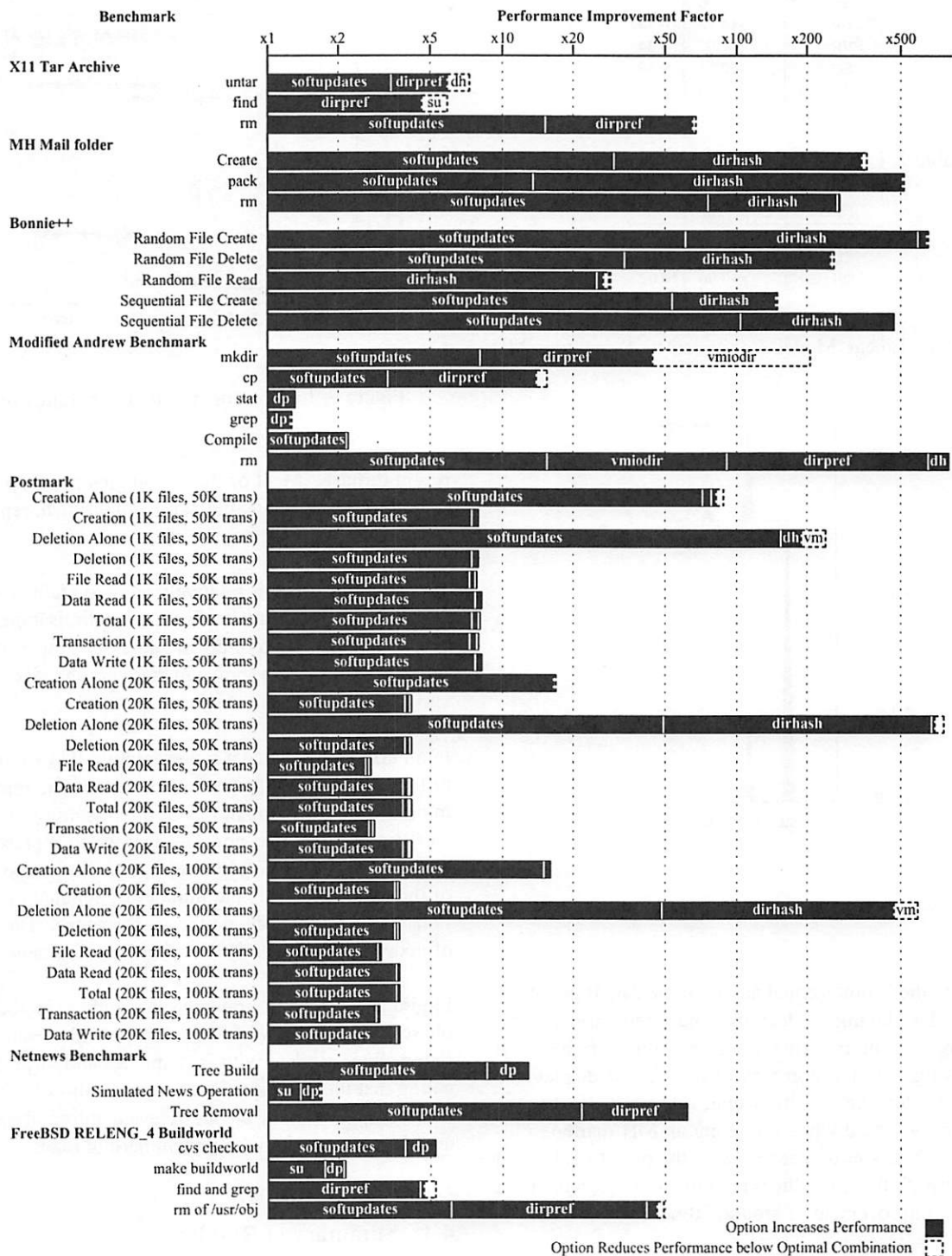


Figure 10: Summary of Results by Benchmark. Each bar shows the improvement factor for a single benchmark, and a breakdown of that gain among the different optimisations. Only the uppermost path across the improvement lattice is used, so larger gains appear on the left of the figure. For some tests, there are combinations of optimisations that result in better performance than having all options enabled. This gain is displayed as dashed regions of the bars; the end of the black region is the improvement factor with all optimisations on and the end of the whole bar is the gain for the best combination.

Of the tests significantly affected by soft updates, most saw the elapsed time reduced by a factor of more than 2 with soft updates alone. Even benchmarks where the metadata changes are small (such as Bonnie++'s sequential data output tests) saw small improvements of around 1%.

Some of the read-only tests were slightly adversely affected by soft updates (Andrew's `grep` 0.7%, Andrew's `stat` 2.4%). This is most likely attributable to delayed writes from other phases of the tests. In a real-world read-only situation this should not be a problem, as there will be no writes to interact with reading.

As expected, `dirpref` shows up as an important factor in benchmarks involving directory traversal: Andrew, Buildworld, Netnews and X11 Archive. For the majority of these tests `dirpref` improves times by 10%–20%. Of all the individual tests within these benchmarks, the only one that is not noticeably affected is the compile time in the Andrew benchmark.

Though it involves no directory traversal, the performance of Postmark with 10,000 files does seem to be very weakly dependent on `dirpref`. The most likely explanation for this is that the directory used by Postmark resides in a different cylinder group when `dirpref` is used and the existing state of the cylinder group has some small effect.

The impact of `vmiodir` was much harder to spot. On both the Andrew compile phase, and the build and remove phases of Buildworld, there was an improvement of around 2%. This is worthy of note as buildworld was used as a benchmark when `vmiodir` was assessed.

`Vmiodir` and `dirpref` have an interesting interaction on the `grep` and remove phases of the Andrew benchmark. Both produced similar-sized effects and the effects show significant overlap. This must correspond to `vmiodir` and `dirpref` optimising the same caching issue associated with these tasks.

As expected, `dirhash` shows its effect in the benchmarks involving large directories. On the Folder benchmark, `dirhash` takes soft updates's already impressive figures ($\times 29$, $\times 13$ and $\times 77$ for create, pack and remove respectively) and improves them even further ($\times 12$, $\times 38$ and $\times 3$ respectively). However, without soft updates, `dirhash`'s improvements are often lost in the noise. This may be because `dirhash` saves CPU time and writes may be overlapped with the use of the CPU when soft updates is not present.

There are improvements too for Bonnie++'s large directory tests showing increased rates and decreased %CPU usage. The improvements for randomly creating/reading/deleting files and sequentially creating/deleting files are quite clear. Only Bonnie++'s sequential file reads on a large directory failed to show clear improvements.

Some improvements were also visible in Postmark. The improvements were more pronounced with 20,000 files than with 10,000. When combined with soft updates, overall gains were about 1%–2%. `Dirhash` presumably would show larger benefits for larger file sets.

5 Conclusion

We have described soft updates, `dirpref`, `vmiodir` and `dirhash` and studied their effect on a variety of benchmarks. The micro-benchmarks focusing on individual tasks were often improved by two orders of magnitude using combinations of these optimisations.

The Postmark, Netnews and Buildworld benchmarks are most representative of real-world workloads. Although they do not show the sort of outlandish improvements seen in the earlier micro-benchmarks, they still show large improvements over unoptimised FFS.

Many of these optimisations trade memory for speed. It would be interesting to study how reducing the memory available to the system changes its performance.

We also studied `dirhash`'s design and implementation in detail. Given the constraint of not changing the on-disk filesystem format, `dirhash` seems to provide constant-time directory operations in exchange for a relatively small amount of memory. Whereas `dirhash` might not be the ideal directory indexing system, it does offer a way out for those working with large directories on FFS.

References

- [1] S. Best. JFS overview, January 2000, <http://www-106.ibm.com/developerworks/library/jfs.html>.
- [2] Russell Coker. The bonnie++ benchmark, 1999, <http://www.coker.com.au/bonnie++/>.

- [3] ISC et al. INN: InterNetNews, <http://www.isc.org/products/INN/>.
- [4] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation*, pages 49–60, Monterey, CA, USA, 14–17 1994. <http://citeseer.nj.nec.com/ganger94metadata.html>.
- [5] J. Katcher. Postmark: A new file system benchmark, October 1997, http://www.netapp.com/tech_library/3022.html.
- [6] Donald E. Knuth. *The Art of Computer Programming: Volume 3 Sorting and Searching*. Addison Wesley, second edition, 1998.
- [7] Marshall Kirk McKusick. Running fsck in the background. In *Usenix BSDCon 2002 Conference Proceedings*, pages 55–64, February 2002, <http://www.usenix.org/publications/library/proceedings/bsdcon02/mckusick.html>.
- [8] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. pages 1–17, 1999, <http://citeseer.nj.nec.com/mckusick99soft.html>.
- [9] Landon Curt Noll. Fowler / Noll / Vo (FNV) hash, <http://www.isthe.com/chongo/tech/comp/fnv/>.
- [10] Grigoriy Orlov. Directory allocation algorithm for FFS, 2000, <http://www.ptci.ru/gluk/dirpref/old/dirpref.html>.
- [11] Daniel Phillips. A directory index for Ext2. In *Proceedings of the Fifth Annual Linux Showcase and Conference*, November 2001, <http://www.linuxshowcase.org/phillips.html>.
- [12] B. Rakitzis and A. Watson. Accelerated performance for large directories, http://www.netapp.com/tech_library/3006.html.
- [13] H. Reiser. ReiserFS, 2001, http://www.namesys.com/res_whol.shtml.
- [14] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000, <http://www.usenix.org/publications/library/proceedings/usenix2000/general/seltzer.html>.
- [15] K. Swartz. The brave little toaster meets usenet, October 1996, <http://www.usenix.org/publications/library/proceedings/lisa96/kls.html>.
- [16] A. Sweeney, D. Doucette, C. Anderson W. Hu, M. Nishimoto, and G. Peck. Scalability in the XFS file system, January 1996, <http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html>.
- [17] Various. The XFree86 Project, Inc., <http://www.xfree86.org/>.

Results

The complete set of scripts, code, tables and graphs are available at <http://www.cnri.dit.ie/~dwmalone/ffsopt>.

Acknowledgments

The authors would like to thank: Robert Watson for his comments and advice; Chris Demetriou and Erez Zadok for providing us with last-minute shepherding; Cathal Walsh for advice on linear models. Some of the work in this paper was developed or tested at the School of Mathematics, Trinity College Dublin where both authors are members of the system administration team.

Filesystem Performance and Scalability in Linux 2.4.17

Ray Bryant, *SGI*, raybry@sgi.com

Ruth Forester, *IBM LTC*, rsf@us.ibm.com

John Hawkes, *SGI*, hawkes@sgi.com

Abstract

The Linux kernel is unique in that it supports a wide variety of high-quality filesystems. For server systems, the most commonly used are Ext2, Ext3, ReiserFS, XFS and JFS. This paper compares the performance of these filesystems using Linux 2.4.17 and three benchmarks: *pgmeter*, an open source implementation of the Intel Iometer benchmark; *filemark* (a version of *postmark*); and *AIM Benchmark Suite VII*. The benchmarks were run on three different systems ranging in size from a contemporary single-user workstation to a 28-processor ccNUMA machine. Although the best-performing filesystem varies depending on the benchmark and system used, some larger trends are evident in the data. On the smaller systems, the best-performing file system is often Ext2, Ext3 or ReiserFS. For the larger systems and higher loads, XFS can provide the best overall performance.

1 Introduction

One of the advantages of open source software is the tremendous number of choices available to users of such software. For example, the filesystem menu of the configuration page for the Linux 2.4.17 kernel includes 28 entries, and this does not include the network filesystems. Although a number of these filesystems are special purpose or for compatibility with other operating systems, the fact remains that there are a wide variety of filesystems available for use with Linux.

Given this large list of filesystems, a Linux user might ask the reasonable question: "Which filesystem should I use?" In many cases the answer to this question depends on functional or ease-of-use issues; in other cases performance is a deciding factor. The goal of this paper is to provide a partial answer to this question by comparing the performance of five of the more popular filesystems available under Linux: Ext2, Ext3, ReiserFS, XFS, and JFS.

As is the case with any benchmark study, we must emphasize that the performance results reported here reflect measurements taken at a particular point in time, with a particular set of machines, with a particular Linux kernel (in our case 2.4.17), using a particular set of benchmarks and parameters, and only truly compare filesystem performance under these restrictive circumstances. Over time, we expect each filesystem discussed herein to continue to be developed, and we expect that the relative orderings of filesystem performance discussed here will change.

Nevertheless, we believe that our benchmarking study does have value in that trends that are identified here can provide broad guidance to individuals who know

the characteristics of their system's workloads and who have the ability to choose which filesystem to deploy. It is also possible that these studies may uncover performance problem areas that the filesystem developers can focus on to improve the performance of their filesystems, thus helping to advance Linux kernel development.

We are aware of no other filesystem benchmarking study with as large a scope and rigor as the one we present in this paper. The best other results we are aware of are the benchmark pages at the ReiserFS site [ReiserFS] and von Hagen's benchmark studies [vonHagen]. However, those studies are only for small, uniprocessor systems. This paper is the only one we are aware of that also examines larger systems.

In the remainder of this paper, we first provide a brief overview of the filesystems we are testing. We then describe the benchmarks we will use in this study. Next, we describe the machines we used to run the benchmarks and the rules that defined how the benchmarks were run. Finally, we present the results of our experiments and discuss the implications and trends these experiments have demonstrated.

2 Filesystem Descriptions

In this section we provide a brief overview and history of the filesystems tested in this paper. Further details about these file systems can be found elsewhere [vonHagen, Galli].

We assume that the reader is familiar with basic concepts of Linux filesystems such as the filesystem *buffer cache* and *journaling* filesystems [Bovet, vonHagen].

Version information for the filesystems and supporting tools used in this paper are given in the Appendix.

2.1 Ext2

Designed by Wayne Davidson (with Stephen Tweedie and Theodore Ts'o) as an enhancement of the *ext* filesystem from Remy Card, Ext2 [Ext2] is the standard Linux filesystem. We include Ext2 in our tests because it provides a baseline of performance that is familiar to many users.

2.2 Ext3

An enhancement of Ext2 developed by Stephen Tweedie [Tweedie], Ext3 uses the same disk format and data structures as Ext2, but in addition supports journaling. This makes conversion from Ext2 to Ext3 extremely easy—no data migration or filesystem reformatting is required. All that one needs to do is to install an Ext3-capable kernel, use the *tune2fs* program to create a journal, and remount the file system.

Ext3 is the default filesystem for Red Hat 7.2 [Ext3RH] and has been included in the standard Linux kernel since 2.4.13. It is available with many Linux distributions.

Ext3 is block based, with sequential filename directory search. Ext3 supports three journaling modes to allow the user to tradeoff performance and integrity guarantees. The default journaling mode, *data=ordered*, guarantees consistent writing of the metadata, descriptor and header blocks and provides good performance; *data=writeback* provides a somewhat lower data-integrity guarantee in favor of better performance (old data can be present in a file after a crash); and *data=journal* provides both metadata and data journaling.

Which of these modes has the best performance varies according to workload [Tso]. One would normally expect that the *data=journal* mode would perform significantly slower than the other modes due to the additional overhead of logging data changes. However, certain workloads have been found to run faster with *data=journal* (especially when reads and writes are to and from the same disks), but the reasons for this are not yet fully understood [Robbins].

In this paper, we report results for *data=writeback* and *data=ordered*. We have not included results with *data=journal* since this is a specialized mode that is not supported by the other journaling filesystems tested here.

2.3 ReiserFS

Developed by Hans Reiser, ReiserFS [ReiserFS] has been part of the standard Linux kernel since 2.4.1 and it is available with many Linux distributions. ReiserFS supports metadata journaling, and it is especially noted for its excellent small-file performance. ReiserFS uses B* Balanced Trees to organize directories, files, and data. This provides fast directory lookups and fast delete operations. Other performance features include support for sparse files and dynamic disk inode allocation.

ReiserFS supports a space-saving option called *tail-packing* that packs small files into the leaves of the B* Tree. However, this option has a performance cost and most benchmarks are run with this feature disabled. In this paper, we follow this practice and report only results with *notail*.

2.4 XFS

Based on SGI's Irix XFS filesystem technology [XFSirix], the XFS port to Linux, version 1.0, was released May, 2001 [XFS]. A large number of support tools are also distributed with XFS [vonHagen, p. 165-167].

XFS is a journaling filesystem that supports metadata journaling. XFS uses allocation groups and extent-based allocations to improve locality of data on disk. This results in improved performance, particularly for large sequential transfers. Performance features include asynchronous write ahead logging (similar to Ext2 with *data=writeback*), using balanced binary trees for most filesystem metadata, delayed allocation [XFS2000], dynamic disk inode allocation, support for sparse files, space preallocation, and coalescing on deletion. One shortcoming can be the poor file deletion performance, which is constrained by synchronous disk writes¹.

For this paper, we followed the recommendations of the FAQ [XFS] and mounted XFS filesystems using the options *-o logbufs=8,osyncisdsync*; *logbufs* specifies the number of log buffers kept in memory, which could improve performance²; *osyncisdsync* specifies that

¹ Asynchronous deletes have been added to more recent versions of XFS (beginning with Linux 2.4.18); however, that version was not used in this paper, since 2.4.18 was not available on all of the systems used in our tests.

² We ran the benchmarks on the "large" system (Section 4) with and without *logbufs=8*, with no discernable performance differences.

O_SYNC is treated as O_DSYNC (data-sync only), the default behavior on Ext2 [XFS].

At the present time XFS is not part of the standard Linux kernel; patchsets for recent kernels are available [XFS], and it has been included in recent versions of the Mandrake distribution [Mandrake].

2.5 JFS

IBM's JFS [JFS], which originated on AIX, was then ported to OS/2, then back to AIX and from there was ported to Linux [vonHagen, p. 106]. It has the advantage that the code has undergone extensive testing under other operating systems. JFS technical features include extent-based storage allocation, variable block sizes (although only 4096 byte blocks are currently supported under Linux), dynamic disk inode allocation, and sparse and dense file support. JFS is a journaling filesystem that supports metadata logging.

JFS is not currently supported on systems where the page size exceeds the filesystem block size, so JFS was not included in our "large"-system benchmarks since the default 16 KB page size on IA64 exceeds the 4 KB JFS block size [Best]. (See Section 4 for description of our "large" system.)

Although JFS is not part of the standard Linux kernel, patchsets for recent kernels are available [JFS], and JFS has been incorporated into recent Mandrake distributions [Mandrake].

3 Benchmark Descriptions

The benchmarks we have chosen for this paper are *pgmeter*, *filemark*, and the *AIM Benchmark Suite VII*.

In a previous paper [Pgmeter], we evaluated many of the other filesystem benchmarks commonly in use on Linux today. These included *Bonnie* [Bonnie], *dbench* [Dbench], and the ever-popular kernel-compile benchmark as well as several others. The advantages of the benchmarks used in this paper over these other benchmarks are that they:

- 1) Support a wide variety of workloads instead of the specific workload implemented by *Bonnie*.
- 2) Are not trace driven, (unlike *dbench*) so that they are readily scalable from small to very large systems.
- 3) Produce significant load on underlying filesystems, unlike the kernel-compile workload that is more often than not CPU bound.

We selected the benchmarks for this paper based in part on our previous work and because we believe these benchmarks represent three interesting and illuminating facets of filesystem performance: I/O throughput, file

accesses, and overall system performance; they scale well on larger multiprocessor platforms; and they report accurate, repeatable results.

3.1 Pgmeter

Pgmeter is a file-I/O benchmark that measures the rate at which data can be transferred to/from an existing file according to flexible, synthetic workload description. *Pgmeter* is patterned after the Intel *Iometer* benchmark [Iometer]. In previous work we demonstrated that *pgmeter* creates workloads that have the same performance characteristics as those of *Iometer* [Pgmeter].

The workload descriptions used here are based on those distributed with *Iometer*:

- Sequential read and write tests with fixed record sizes.
- The *Web Server* workload, which is a mixture of 512 byte to 512 KB transfers. All transfers in this workload are read operations; each transfer begins at a randomly selected point in the data file.
- The *File Server* workload, which is a mixture of 512 byte to 64 KB transfers. Of these transfers, 80% are read operations and 20% are write operations. Each operation begins at a randomly selected point in the file.
- An *8K OLTP* workload, defined as 8 KB transfers of which 67% are reads and 33% are writes. Each transfer begins at a randomly selected point in the file. This workload is intended to represent an online transaction processing workload.

Additional details of the Web Server and File Server workloads are provided in the Appendix.

We selected these workloads with a goal of covering a broad range of representative workloads while keeping the total number of tests small. For example, we did not include a purely random, read-only, fixed-record size workload, since the Web Server workload is also a random, read-only workload; we felt that many of the characteristics of the purely random workload would show up in the Web Server tests. Similarly, we believed that the fixed record size, random access benchmark would be covered (in part) by the 8K OLTP workload.

The key statistics reported by *pgmeter* are the filesystem I/O bandwidth, measured in MB/s, and the filesystem operations count, measured in ops/s. In this paper we report only the ops/s statistic; the MB/s results show similar results to those presented here.

Pgmeter includes the following features to increase the accuracy of the test results:

- 1) Each test begins by unmounting and remounting the target filesystem as a method to flush the filesystem buffer cache. Thus at the start of each test, the filesystem buffer cache is empty.
- 2) A warm-up period can be configured to run at the start of each test. During the warm-up period the full benchmark is run, but statistics collection does not begin until the end of the warm-up period.

The purpose of the warm-up period is to exclude measurement of transient startup effects. An example of this would be the abnormally high write-rate that occurs in a sequential write test until the filesystem buffer cache fills up and the filesystem has to actually start writing data to disk.

3.2 Filemark

Filemark is our version of the *Postmark* benchmark [PostMark]. *Postmark* is a filesystem-operation-intensive benchmark. *Postmark* execution consists of three phases:

- 1) Creation Phase. During this phase, *postmark* creates a specified number of files (See Section 5, "Benchmark Run Rules" for details). These files are randomly distributed among a number of subdirectories of the target directory.
- 2) Transaction Phase. During this phase, *postmark* executes a number of "transactions" against the set of files created in the Creation Phase. Each transaction consists of the following two steps: (i) Choose a file at random, then according to a second random choice, either read the entire file or append to the file. (ii) According to a random choice either create a new file or delete an existing file.
- 3) Deletion Phase. During this phase, the files remaining in the file set are deleted.

Our version of *postmark*, which we call *filemark*, includes the following six enhancements to *postmark 1.5*:

- 1) *Filemark* is multithreaded whereas *postmark 1.5* is single-threaded.³ This allows us to scale up the *filemark* workload to provide a heavier and more realistic load on the server machines we are testing.

³ While writing this paper, we became aware of a multithreaded version of *postmark*, version 1.99. [Katcher]. Due to time constraints we have continued to use *filemark* in this paper rather than convert to this new version of *postmark*.

- 2) *Filemark* uses higher precision timing services than does *postmark*. *Filemark* uses *gettimeofday()* instead of *time()*. The former is nominally accurate to the nearest microsecond while the latter is accurate only to the nearest second.
- 3) *Filemark* supports repeated *Transaction Phases* following a single *Creation Phase*. This was incorporated to significantly reduce the amount of time required to get multiple repeated trials suitable for confidence interval generation. Although each transaction phase uses basically the same set of created files, the transaction rate was assumed not to depend that much on the particular set of files created. Each transaction phase begins with its own warm-up period and ends by unmounting and remounting the filesystem containing the target directory. In this way, each transaction phase is as much of an independent trial as possible, and standard techniques for confidence interval calculation should apply.
- 4) *Postmark* only allows the *bias read* and *bias create* probabilities (these are the probability of choosing read over append, or create over delete, in steps (2)(i) and (2)(ii) above) to be specified to the nearest 10 percent; *filemark* allows these probabilities to be specified to the nearest 1 percent.
- 5) Code has been added to *filemark* to allow the user to request that the Deletion Phase not be run at all. This was included to speed the execution time of the entire test. Our experience is that the file Deletion Rate as reported by *filemark* (or by *postmark*) has such high variation as to be almost meaningless. Also, for our run rules (see Section 5), we always rebuilt (reformatted) the filesystem after a *filemark* run, so we did not need to run the Deletion Phase.

3.3 AIM7

The *AIM Benchmark Suite VII*, referred to here as *AIM7*, has been widely used for more than a decade by many Unix computer system vendors. AIM Technology, Inc. originally developed and licensed the AIM Benchmark suites. Caldera International, Inc., has acquired the AIM Benchmark license and has recently released the sources for *Suite VII* and *Suite IX* under the GPL [AIM7].

AIM7 is a C-language program that forks multiple processes (called *tasks* in *AIM7*), each of which concurrently executes a common, randomly-ordered set of subtests called *jobs*. Each of the 53 kinds of jobs exercises a particular facet of system functionality, such as disk-file operations, process creation, user virtual memory operations, pipe I/O, and compute-bound

arithmetic loops. *AIM7* includes disk subtests for sequential reads, sequential writes, random reads, random writes, and random mixed reads and writes.

An *AIM7* run consists of a series of subruns with the number of tasks, N , being increased after the end of each subrun. Each subrun continues until each task completes the common set of jobs. The performance metric, “Jobs completed per minute”, is reported for each subrun. The result of the entire *AIM7* run is a table showing the performance metric versus the number of tasks, N .

Typically, as N increases, the jobs per minute metric increases to a peak value as CPU idle time gets used for real work and as economies of scale continue to succeed; thereafter it declines due to software and hardware bottlenecks. This peak throughput value provides the primary metric of interest. We report the peak throughput achieved with each filesystem tested as our *AIM7* statistic for that filesystem.

3.4 Benchmark Summary

To reiterate, *pgmeter* measures the rate at which data can be transferred to or from an existing file. It is intended to model the behavior of large applications as they read and update their associated disk files. *Filemark* is intended to model the kind of operations that a file server or mail server might execute against a filesystem. By choosing these two benchmarks, we have thus chosen opposite ends of the spectrum of possible filesystem workloads. Whereas *pgmeter* measures the filesystem’s ability to transfer data under a wide variety of workloads, *filemark* measures the filesystem’s ability to create and update a number of small files. We feel that real-user applications lie somewhere between these two extremes and that our benchmarks should therefore encompass the actual workloads of many real-user programs.

AIM7 serves as a multifaceted system-level benchmark that exercises more than just filesystem activity. We make no claim that an *AIM7* workload represents a “real world” job mix. Rather, we claim that *AIM7* imposes a workload that shows how a particular Unix system scales under the stress of an ever-increasing load. We believe that since the *AIM7* workload includes a mix of both CPU-bound and I/O-bound jobs, that *AIM7* provides an estimate of overall system throughput that is not provided by *pgmeter* or *filemark*.

4 Experimental Testbeds

In this paper we have executed our benchmarks using three systems:

- 1) The *small* system. This is a single-processor 1.7 GHZ Pentium 4 with 512MB of memory and an 80 GB IDE disk. For the experiments of this paper, this machine was booted with either 128MB or 512MB of RAM.
- 2) The *medium* system. This is a 4-CPU 700 MHZ Pentium III Xeon system with 5 SCSI disks, 8 GB each. For the experiments of this paper, this system was booted with 900 MB of RAM.
- 3) The *large* system. This is an SGI prototype of a ccNUMA machine using the Intel IPF processor. Currently based on the Itanium and a system interconnect similar to that of the SGI Origin 3000 [SGIorigin], this machine allows us to run a modified Linux 2.4.17 kernel with up to 28 processors, 16 GB of main memory, and 10 Fibre Channel controllers attached to a total of 120 disk drives.

The above set of machines provides a sampling from a wide variety of machines used to run Linux today. Although it may still be that Linux is most commonly run on uniprocessor machines, the 4-CPU system represents a “sweet spot” for Linux mid-range servers and thus is an important system to include. Similarly, while the 28-processor SGI machine reflects a class of system that is rarely found in the Linux world today, we believe that such systems will be more commonplace in the next few years. Moreover, benchmarking on such machines tends to exaggerate and thus more clearly expose performance problems that are less apparent on smaller hardware configurations.

5 Benchmark Run Rules

Filesystem benchmarking requires careful setup [Tang]. An issue one must often contend with is how to defeat the effects of the filesystem buffer cache. Without careful experimental design, all of the filesystem requests could be satisfied in the cache and no disk activity would occur. A common way to avoid this problem is to use a total file size that exceeds the amount of main memory available on the system.

Another approach is to use a file-access mode that bypasses the filesystem buffer cache, such as `O_DIRECT`. We chose to not use `O_DIRECT` for this paper in order to focus on the default filesystem behavior that most users will encounter.

The second issue that one must address is estimating the accuracy of the results of the test. In our experience, filesystem benchmarks are notorious for being

nonrepeatable, bimodal, and full of hysteresis effects, making it a challenge to get consistent results.

In this paper, we estimate the accuracy of the tests using 95% confidence intervals calculated using a Student's-T statistic. Such statistics usually require 10 or more independent trials to get a reasonably tight confidence interval. However, the large number of filesystems examined in this paper, the number of tests per filesystem and the running times of the experiments meant that running the test multiple times was not feasible. Our solution was to use intermediate test results to create quasi-independent observations of the test statistics, and then to generate confidence intervals based on these observations.

Finally, one must be aware of internal bottlenecks within the hardware I/O subsystem that can otherwise skew results. For example, we run all of our benchmarks (except for the ones on the small system) using multiple physical disks. We do this because this improves parallelism and allows higher I/O rates to be achieved; this results in a higher load being placed on the filesystem that we are testing. Additionally, this is sometimes necessary to avoid excessive disk-head movement that would otherwise invalidate the test. For example, if multiple sequential read tests are executed at the same time on a single disk drive, the head movement pattern may be nearly indistinguishable from that of a random access workload, and the resulting transfer rates will be much slower than otherwise might be achieved.

5.1 Pgmeter

For the *pgmeter* tests, the total size of the test file(s) used was always chosen to be significantly larger than the main memory of the system. For the sequential read and write tests, this implies that each user-level I/O request results in a disk request once the filesystem buffer cache is filled (except for the effect of read ahead in the Linux kernel). We used a warm-up period of 30 seconds. (This was based on examining the *pgmeter* output when no warm-up period was specified and making a judgment about how long it took to get the system to steady-state behavior.) We used an experiment runtime of 5 minutes for each *pgmeter* test case.

For the random access tests, the effect of the filesystem cache is that certain requests will be satisfied out of cache. At the start of the test, the cache is empty (since we remount the target filesystems at the start of the test), and as the test progresses more and more of the target file is read into memory. Thus, the effect of the filesystem cache in the random access tests is to cause the filesystem data transfer rate to appear to increase as

the test continues to run. To minimize this effect, we chose a file size large enough that only a small fraction of the file is read into memory during the test. Examination of the *pgmeter* output also shows that the data rate in each 10-second interval of the test was very nearly constant in all of the random access tests conducted for this paper.

On the small system, we ran *pgmeter* against a single 1 GB file, with the sequential read and write cases using an I/O size of 8 KB. We ran trials with 1, 4, 16, and 64 outstanding I/O requests against this file. Main memory on the system was 512MB.

On the medium system, we ran *pgmeter* using 4 files, each 2 GB in size, with each file being on a distinct physical disk. The I/O size for the sequential read and write tests was 64 KB. We ran trials with 1, 4, 16, and 64 outstanding I/O requests against each of the files. Main memory on the system was 900MB.

On the large system, we ran *pgmeter* using 28 files, each 2 GB in size, with each file being on a distinct physical disk. (We also report some results using 112 files.) The I/O size for the sequential read and write tests was 64 KB. We ran trials with 1, 2, 4, and 8 outstanding I/O requests against each of the 28 files. These numbers were reduced from the small and medium cases due to the large (total) number of processes that would otherwise be created in this case. The main memory size was 16 GB.

We calculated confidence intervals for *pgmeter* as follows. Interval statistics were recorded by *pgmeter* every 10 seconds. That is, we calculated the MB/s and ops/s for every 10 seconds of the run. The statistic reported by the benchmark itself is the average of all of the interval statistics over the run. We used the interval statistics to calculate an estimate of the mean and 95% confidence intervals for the mean.

5.2 Filemark

We ran *filemark* for 1, 8, 64, and 128 threads for each of the five filesystems available for testing on IA32. The filesystems were recreated using *mkfs* at the start of each test for each number of threads. The system was not rebooted either at the start of a new filesystem test nor at the start of a new number of threads test. As previously discussed, only one Creation Phase was run per case and then the repeated Transaction Phases were run without running another Creation Phase. No Deletion Phase was run. Instead, the filesystem was rebuilt after each *filemark* run.

After the benchmark runs completed, the first transaction was discarded and we used the remaining transac-

tion rates to calculate the mean transaction rate and a 95% confidence interval for the mean. Our experience was that the first trial was often an outlier, due to startup effects. Additional trials were conducted for any test where the confidence interval half-width was larger than 10% of the mean.

It was only through this approach that we are able to obtain meaningful statistics from the *filemark* benchmark. Random variations between tests were otherwise so high as to make the measured statistics meaningless. Repeating the entire series of tests enough times to generate confidence intervals was not feasible given the extended run times of each series of tests (16–18 hours for a complete run over all file systems).

Of course, this approach did not allow us to calculate confidence intervals for the Creation Rate; only by using repeated trials could we provide that data. However, we regard the most important statistic reported by *filemark* to be the Transaction Rate, since this represents the rate at which a file or mail server would be able to process requests. This is the statistic reported in this paper and we do not include Creation Rate data.

Finally, it was necessary to make the file creation and deletion probabilities equal in order to keep the number of files nearly constant. (In *filemark*, this is done by setting *bias create* to 50.) Without this change, the results of each trial were not taken from a stationary distribution and the confidence interval calculation was not statistically valid.

In all of our *filemark* runs, *bias read* was 75 (this is the probability expressed as a percent of reading versus appending to a file in part (i) of a *filemark* transaction) and *bias create* was 50 (this is the probability of creating versus deleting a file in part (ii) of a *filemark* transaction).

For the small system, we ran *filemark* with the following parameters: 1 target directory with 10 subdirectories, 10,000 total files, 30-second warm-up per transaction phase, 2-minute transaction phases, 12 transaction phases per run. The number of subdirectories was chosen, in part, from input received from the ReiserFS mailing list [ReiserFS]. Because the file set here is relatively small, the system was booted with only 128MB of memory (instead of 512 MB).

For the medium system, we ran *filemark* with 4 target directories, each on a different physical disk, 2000 subdirectories per target directory, 100,000 total files, 30-second warm-up per transaction phase, 2-minute transaction phases, and 6, 12, or 24 transaction phases per run, depending on what was required to get the confidence interval half-width to be smaller than 10%

of the mean. As before, we discarded the first trial of the series before the confidence intervals were calculated.

In addition to the 10,000 and 100,000 file cases, we tested two different file-size distributions with *filemark*. In the first case, which we will refer to as the *small-file workload*, file sizes were chosen uniformly between from 512 bytes to 9.77 KB (this is the *postmark* default) and the read and write I/O sizes were 512 bytes. In what we will refer to as the *large-file workload*, file sizes ranged from 4 KB to 16 KB, and the read and write I/O sizes were 4 KB. Since the block size for all of the filesystems used in this paper is also 4 KB, we designed the large-file case to avoid filesystem read-before-write overheads.

The total size of the file set for the small-file workload is roughly 50 MB for the 10,000-file case and 500 MB for the 100,000-file case. For the big-file workload the file set has size 100 MB in the 10,000-file case and approximately 1 GB in the 100,000-file case. The large-file case is thus clearly larger than the main memory in the small system 128 MB case and the medium system's 900 MB case, while for the small-file case the entire file set could fit into the filesystem buffer cache. This was one reason for running both the small and large-file cases. The other reason was to examine the effect of the request size change from 512 bytes to 4 KB.⁴

We did not run *filemark* for the large system because it is a small-server benchmark. Even for the medium system, the entire file-set consists of only 500 MB of files. The large system has 16 GB of RAM. We have thus far been unable to scale up *filemark* sufficiently that it would reliably generate more than 16 GB of files.

5.3 AIM7

Although *AIM7* can be executed against a single disk drive, our experience is that the filesystem jobs are disk-transaction-rate constrained. With too few disk drives available, the *AIM7* test will be almost completely I/O bound. The small and medium systems used for our tests were thus too small to run a meaningful *AIM7* test. For our large system we have determined that 120 disk drives is more than sufficient to produce a compute-bound *AIM7* test.

⁴ We also ran the same benchmark on the small system with larger memory (512 MB instead of 128 MB) and on the medium system with smaller memory (512 MB instead of 900 MB) and found no significant change in the results from those reported in Section 6.

For our *AIM7* benchmark runs, we chose the *AIM7 workload.shared* workload. *AIM7* documentation [AIM7] describes this job mix as a “Multiuser Shared System Mix”.

We repeated the benchmark runs on the 28 CPU large system with the system booted with 2, 4, 8, 16, 24 and 28 processors. At each CPU count we executed the *AIM7* benchmark against each of the four filesystem types (JFS is not currently available on IA64); we recorded the maximum jobs completed per minute statistic. We rebooted the system before each test run. Each of the 120 disk drives used was formatted with a single filesystem partition.

We executed only a single trial of each *AIM7* benchmark for each filesystem and CPU configuration. This is due to the length of each *AIM7* run, requiring one to ten hours to reach a peak throughput. A single try should be sufficient because each *AIM7* run actually consists of a number of subruns (one for each workload level), and examination of the graph of “jobs completed/min” versus the number of simultaneous runs would allow us to detect and discard anomalous results.

6 Results

We concentrate here on where the benchmarks show significant differences between the filesystems. A significant difference between filesystems is assumed to exist if the 95%-confidence intervals for the two filesystems do not overlap. Additionally, we typically require a difference of at least 10% in the mean values for a difference to be considered significant.

6.1 Small System

On the small system, the filesystems all produced about the same results on all of the *pgmeter* tests (the 95% confidence intervals for the results overlapped). Comparison of the *pgmeter* statistics for the sequential read tests with those of *iostat* shows that each of the filesystems drives the disk at nearly its maximum rate of around 2200 I/O operations/s. (*hdparm -t -T* shows that the disk can maintain 17.4 MB/s; this would correspond to around 2227 I/O operations/s for an 8 KB request size). We conjecture that these *pgmeter* tests are all disk-I/O limited and thus produce equivalent results for all filesystems.

The *filemark* results for the small system do show significant differences among the tested filesystems. Figure 1 shows the results for the small file workload. In this figure, ReiserFS, Ext2, and Ext3 are in the upper group and XFS and JFS are in the lower group. ReiserFS scales well with increasing number of threads, and

it provides significantly better performance than Ext2 and Ext3. Neither XFS nor JFS improve very much as the number of threads increases, and XFS and JFS provide performance at basically the same level. These results are consistent with the *Bonnie* tests for random seeks as reported by von Hagen [vonHagen, p. 240].

We surmise that the poor performance of XFS and JFS in this environment is due to the large-system heritages of these filesystems; optimizations appropriate for a larger system may not be beneficial in a small-system environment.

Figure 2 shows the same experiment with the big-file workload. In this test, Ext2 and Ext3 begin with higher transaction rates for one thread; as the number of threads grows, ReiserFS performance increases until it nearly reaches the rates of Ext2 and Ext3 with *data=ordered*. Ext3 with *data=writeback* is significantly slower than the other filesystems at the larger number of threads. These results are consistent with those available on the ReiserFS web site [ReiserFS], where ReiserFS performance is often the best for smaller file sizes.

XFS and JFS still perform the slowest; but in this case there is a clearer distinction between XFS and JFS.

The trends exposed by this set of experiments on the small system appear to be the following:

- 1) For *file-I/O intensive workloads* on the small system, all of the filesystems provide equivalent performance.
- 2) For small files, where the I/O request size is less than the filesystem block size, and using filesystem-operation intensive workloads, ReiserFS may be fastest.
- 3) For somewhat larger files, where the I/O request size is equal to the filesystem block-size, and using filesystem-operation intensive workloads, Ext2 or Ext3 with *data=ordered* may be fastest.

6.2 Medium System

When we looked at the medium system results, we did find differences among the filesystems when running the *pgmeter* benchmark. Figure 3 shows the results of the *pgmeter Sequential Read 64 KB* workload on the medium system. This graph shows a clear performance advantage for JFS and XFS over the other filesystems, particularly as the number of outstanding I/O's per file increases. It appears that the large-system heritage of these filesystems starts to pay off on the medium system.

This trend did not carry through for the other *pgmeter* workloads, however, where the results for all filesystems

tems were equivalent. In particular, all of the filesystems provided nearly identical performance for the *Web Server* case, indicating that for a read-only workload with variable request sizes, all of the filesystems are equivalent. Figure 4 shows the results of *pgmeter File Server* workload and it indicates that for a mixed read-write workload with a variable request size, XFS and ReiserFS provide marginally less performance than the other filesystems. Figure 5 shows that XFS moves back into the upper group for a mixed read/write workload with constant 8 KB record sizes.

Figure 6 shows the results of the *filemark* benchmark for the medium system and the small-file workload. If we compare this figure to Figure 1 we see that in this case XFS has moved into the upper group. XFS, Ext2, and ReiserFS provide equivalent levels of performance. Examination of the 95% confidence intervals shows that the ReiserFS 64 thread and 128 thread results in this figure are statistically indistinguishable. We therefore do not regard the peak that occurs for ReiserFS at 64 threads in Figure 6 to be significant. Both versions of Ext3 are in the second group for 64 and 128 threads and are equivalent to the top three filesystems for the 1 and 8 thread cases. JFS continues to trail the performance of the other filesystems for this benchmark.

Figure 7 shows the results of the *filemark* benchmark for the medium system and the large-file workload. This graph looks basically the same as the small-file case shown in Figure 6. The corresponding comparison for the small system (see Figures 1 and 2) found significant differences between the small-file and large-file workloads. We have not determined precisely why these two comparisons are different. However, for the small system, large-file workload case, the entire file set fits into memory (100 MB of files and 128 MB of memory). In the medium system, large-file workload case, the entire file set is too large to fit into memory (1 GB of files and 900 MB of memory). This may explain why the differences between Figures 1 and 2 are not also seen between Figures 6 and 7.

The trends exposed by this set of experiments are the following:

- 1) XFS and JFS appear to perform better (in relation to the other filesystems tested) on sequential workloads on multiple processor systems than they did on uniprocessor systems.
- 2) Ext3 appears to have a scaling bottleneck not present in Ext2 (possibly journal related) that keeps its throughput from increasing as the number of threads increase under the *filemark* benchmark.
- 3) In spite of good performance on the sequential read benchmark of *pgmeter*, JFS did not scale well with

either multiple processors or increasing numbers of threads under the *filemark* benchmark.

6.3 Large System

The differences between filesystems are clear when we examine the output for *pgmeter* on the large system. Figures 8, 9, and 10 show results of the *pgmeter* runs for *Sequential Read 64KB*, *Sequential Write 64KB*, and *File Server* respectively. (Recall that JFS was not available for the large system.) Figures 8 and 9 show that XFS provides significantly better performance than the other filesystems for this constant-request-size and regular workload. To continue the previous discussion, we can now observe that this is where the large-system heritage of XFS is truly beneficial.

On the other hand, Figure 6 shows that for a random request size and 100% random access, Ext3 with *data=writeback*, and ReiserFS are the best performing filesystems. Ext3 with *data=ordered* is next, and XFS is last. This suggests that for random access and variable request size workloads, one of these other filesystems might be superior to XFS.

Figure 11 shows the peak jobs-per-minute throughput curves for the *AIM7* benchmark for each tested filesystem type at each CPU count.

Note that the jobs-per-minute y-axis values in Figure 11 are expressed as numbers for each filesystem type relative to the baseline 2-processor-Ext2 value. This is because the large system is a prototype system, and SGI management requested that we not publish absolute *AIM7* performance numbers for this system.

The *AIM7* benchmark indicates that the four filesystems separate into two performance groupings, with Ext2 and XFS performing roughly equivalently, and Ext3 and ReiserFS performing at significantly lower levels. Ext2 and XFS are essentially equivalent performers up to 16 processors. Above 16 processors XFS achieves marginally superior scaling, and at 28 processors XFS reaches a peak throughput about 14% higher than Ext2's peak. Ext3 and ReiserFS perform at about 65% of the level of Ext2 at 2 processors, and neither of these two filesystem types gains much improvement in peak throughput at higher processor counts. Both Ext3 and ReiserFS see a small decrease in peak throughput above 8 processors. A similar small decrease is seen with Ext2 above 16 processors, although overall Ext2 peak throughput is substantially higher than Ext3 and ReiserFS.

We interpret these results as indicating that both Ext3 and ReiserFS are constrained by bottlenecks that are significantly more severe than those that affect Ext2

and XFS. Past experience with this ccNUMA platform leads us to believe that the bottlenecks are primarily highly contended kernel spinlocks.

By adding Lockmeter [Lockmeter] to our 2.4.17 kernel, we can measure spinlock contention. Table 1 shows the percent of all available CPU cycles that were consumed by busy waiting for three spinlocks while running the *AIM7* benchmark with the indicated filesystem. (The data is for the 500 *AIM7* processes case on our large system.) Both Ext3 and ReiserFS are dominated by the so-called *Big Kernel Lock* (BKL), with 85% to 88% of all CPU cycles being consumed by busy waiting for that spinlock. The BKL is still dominant for Ext2, although the *runqueue_lock* (protecting the single global CPU runqueue) now emerges as being almost as significant. For XFS the BKL is even less important, and the *runqueue_lock* is the primary scaling bottleneck.

Filesystem	BKL	runqueue_lock	pagecache_lock
Ext2	30%	25%	5%
ReiserFS	88%	3%	< 1%
Ext3	85%	5%	< 1%
XFS	12%	35%	10%

Table 1. Percent of Total CPU Cycles in Spin Wait for Locks during AIM7 runs on the Large System

The effects of spinlock contention are nonlinear and sometimes surprising. For example, one might imagine that a multiqueue CPU scheduler patch (e.g., the IBM-contributed patch [IBM-MQ]) would eliminate *runqueue_lock* contention and thus gain back those cycles consumed by spinlock waits. For XFS, where the predominant *AIM7* bottleneck in the baseline 2.4.17 kernel is the *runqueue_lock*, when we apply the IBM Multiqueue Scheduler patch, the *AIM7* peak at 28 processors increases to 40% (up from 14%) above the unpatched Ext2 performance. However, for Ext2 with its 30% wasted CPU cycles waiting on the BKL and 25% waiting on the *runqueue_lock*, an unfortunate side-effect of the IBM Multiqueue scheduler is that contention on the BKL increases nonlinearly to a 70% waste of CPU cycles, resulting in a 30% drop in the *AIM7* peak throughput. The lesson here is that the highest contending spinlocks should be attacked first.

This spinlock analysis leads us to conjecture that XFS might perform better than the other filesystems when CPU utilization is high. For example, for the *pgmeter* sequential read and write tests, CPU utilization was

nearly 100% for all filesystems except XFS. Our experience indicates that this is likely due to spinlock contention for those other filesystems. For the cases where CPU utilization is lower, then XFS does not perform as well as the other filesystems (e.g., the File System workload in Figure 10). This conjecture suggests that XFS should perform better on the mixed workload cases if we could increase the load to the point where CPU utilization becomes a limiting factor for the other filesystems.

Figure 12 shows the result of a 112-file test case on our large system. This represents a four-fold increase in system load over the previous 28-file case. For this case, CPU utilization is very high for all of the file systems except XFS, and XFS performs better than the other filesystems.

The trends exposed by the experiments on the large system are the following:

- 1) At the highest loads, all of the filesystems start to become CPU bound. XFS suffers less from this problem than the other filesystems; hence it is able to deliver better service at these higher loads. A significant part of the CPU load for the other filesystems is due to spinlock contention that is higher than that found in XFS.
- 2) In cases where the system is not quite so busy, filesystems other than XFS can provide better performance than XFS does.
- 3) Ext2 and XFS scale the best as the number of processors increase (as measured by the *AIM7* workload). XFS is the only journaling filesystem able to provide the same level of scaling as Ext2.
- 4) Ext3 appears to have internal scaling bottlenecks associated with the Ext3 specific code since Ext2 scales well on *AIM7*, but Ext3 does not.
- 5) ReiserFS has similar scaling problems to those of Ext3.

7 Conclusions

We stated specific conclusions for each of the small, medium and large filesystems in Section 6. In this section we attempt to provide what appear to be performance trends that are true over the entire set of experiments:

- 1) In general, we found the performance of Ext2 to be better than that of Ext3. In most cases, we found the performance of Ext3 with *data=writeback* to be the same as with *data=ordered*. In certain cases, we found Ext3 with *data=ordered* to be faster; in other cases *data=writeback* is faster. On the large system, however, we found performance of *data=writeback* to be even better than that of Ext2

(See Figure 10). It is therefore not clear to us how Ext3 users might be able to predict which journaling mode should be used for their workload.

- 2) For small files (5 KB average size or less) and file-system-operation-intensive workloads, ReiserFS can often provide the best performance.
- 3) For File-I/O intensive workloads like *pgmeter*, all of the filesystems are equivalent on systems like our small system.
- 4) When performance is of primary importance, neither XFS nor JFS are good choices for systems like our small system and filesystem operation-intensive workloads like *filemark*.
- 5) On small SMP systems like our medium system, XFS and JFS are good choices for sequential read workloads.
- 6) JFS appears to have difficulty scaling with SMP or the number of concurrent threads in a filesystem operation-intensive workload like *filemark*.
- 7) XFS currently performs best on systems like our large system under heavy I/O workloads.

8 Future Work

In this paper we have observed behavior without providing a detailed analysis of why the various filesystems perform as they do. This is ongoing work and we expect to provide such insight in future papers on filesystem performance, as well as large-system performance and scalability under the Linux operating system. However, this is a difficult problem, with sometimes surprising results [Robbins].

Finally, nothing in the Linux kernel stays the same for long. Over time, studies such as this one will need to be repeated in order to update the results to new versions of Linux and its filesystems.

9 Software Availability

The *pgmeter* and *filemark* benchmarks are available under the GPL and the Artistic License, respectively at the SourceForge sites *pgmeter.sf.net* and *filemark.sf.net*. *AIM7* is available at Caldera's Web site [AIM7].

10 Acknowledgements

The authors would like to acknowledge the support and assistance of the management at SGI and IBM in the creation of this paper. We gratefully acknowledge the help and assistance of the anonymous referees, as well as the assistance of the shepherd for this paper, Erez Zadok of Stony Brook University. We additionally acknowledge the contributions that VA Linux, Inc. has made through the SourceForge project. Roger Sunshine wrote much of the code for *pgmeter* and we continue to greatly appreciate his contributions.

11 References

- [AIM7] <http://www.caldera.com/developers/community/contrib/aim.html>
- [Best] Steve Best, personal communication, March 4, 2002
- [Bonnie] <http://www.textuality.com/bonnie>
- [Bovet] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly and Associates (2001), ISBN 0-596-00002-2.
- [Dbench] <http://www.samba.org>
- [Ext2] Rémy Card, Theodore Ts'o, and Stephen Tweedie, "Design and Implementation of the Second Extended Filesystem," *Proceedings of the First Dutch International Symposium on Linux*, 1994, ISBN 90-367-0385-9, also available at <http://web.mit.edu/tytso/www/linux/ext2intro.html>.
- [Ext3RH] <http://www.redhat.com/support/wpapers/redhat/ext3/index.html>
- [Galli] Ricardo Galli, "Journal File Systems in Linux," *Upgrade*, Vol. II, No. 6 (December 2001), pp. 50-56, <http://www.upgrade-cepis.org/issues/2001/6/up2-6Galli.pdf>
- [IBM-MQ] <http://lse.sourceforge.net/scheduling>
- [Iometer] <http://developer.intel.com/design/servers/devtools/iometer/index.htm>
- [JFS] <http://oss.software.ibm.com/jfs>
- [Katcher] Jeffrey Katcher, personal communication, February 26, 2002.
- [Lockmeter] <http://oss.sgi.com/projects/lockmeter>
- [Mandrake] <http://www.mandrake.com/en/features.php3>
- [Pgmeter] Ray Bryant, Dave Raddatz, and Roger Sunshine, "Penguinometer: A New File-I/O Benchmark for Linux," *Proceedings of the 5th Annual Linux Showcase and Conference*, Nov 8-10, 2001. <http://www.linuxshowcase.org/tech.html>.
- [PostMark] Jeffrey Katcher, "PostMark a New Filesystem Benchmark," Network Appliance Technical Report TR3022, http://www.netapp.com/tech_library/3022.html.
- [ReiserFS] <http://www.reiserfs.org>
- [Robbins] Daniel Robbins, "Common threads: Advanced filesystem implementer's guide, Part 8," IBM Developer Works, <http://www-106.ibm.com/developerworks/linux/library/l-fs8.html?dwzone=linux>.
- [SGIorigin] <http://www.sgi.com/origin/3000>
- [Tang] Diane Tang and Margo Seltzer, "Lies, Damned Lies, and File System Benchmarks," in VINO: The 1994 Fall Harvest, Harvard Division of Applied Sciences Technical Report tr-34-94, 1994, <ftp://ftp.deas.harvard.edu/techreports/tr-34-94.ps.gz>.

[Tso] Theodore Ts'o, personal communication, March 8, 2002.

[Tweedie] Stephen Tweedie, "Ext3, Journaling Filesystem," Ottawa Linux Symposium 2000, html version available at OLS Transcription Project, <http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>.

[vonHagen] William von Hagen, *Linux Filesystems*, Sams Publishing, 2002. ISBN 0-672-32272-2.

[XFS] <http://oss.sgi.com/projects/xfs>

[XFSirix] Adam Sweeney, Doug Doucette, et al, "Scalability in the XFS File System," *Proceedings of the 1996 Usenix Annual Technical Conference*, San Diego, Cal., (January 22-26, 1996), http://oss.sgi.com/projects/xfs/papers/xfs_usenix

[XFS2000] Jim Mostel, et al, "Porting the SGI XFS File System to Linux," *Proceedings of the 2000 Usenix Annual Technical Conference*, San Diego, Cal., (June 18-23, 2000), <http://www.usenix.org/publications/library/proceedings/usenix2000/freenix/mostek.html>.

Appendix

Filesystem Versions Used

The version of Ext3 used in this paper is the default version available in Linux Kernel 2.4.17. The Ext3 used tools used are from e2fsprogs-1.25 and util-linux-2.11g.

The version of ReiserFS used in this paper is 3.6.25; this is the default version available in Linux kernel 2.4.17. The version of ReiserFS tools used is 3.x.0k_pre11.

The version of XFS used here on our small and medium systems is the 2.4.17 XFS patch available from SGI [XFS] (xfs-2.4.17-all.i386.bz2). A comparable version, developed at SGI for our ccNUMA prototype, is used on our large system.

The version of JFS used here is 1.0.15 for Linux Kernel 2.4.17. The patch files are: jfs-2.4.common-1.0.15-

patch, jfs-2.4.17-1.0.15-patch, and jfsutils-1.0.15.tar.gz, all taken from [JFS].

Web Server Workload

The Web Server workload (as distributed with *Iometer* [Iometer]) is defined by the following parameters:

Request size in bytes	Percent requests this size	Percent read (vs. write)	Percent random (vs. sequential)
512	22	100	100
1024	15	100	100
2048	8	100	100
4096	23	100	100
8192	15	100	100
16384	2	100	100
32768	6	100	100
65536	7	100	100
131072	1	100	100
524288	1	100	100

File Server Workload

The File Server workload (as distributed with *Iometer* [Iometer]) is defined by the following parameters:

Request size in bytes	Percent requests this size	Percent read (vs. write)	Percent random (vs. sequential)
512	10	80	100
1024	5	80	100
2048	5	80	100
4096	60	80	100
8192	2	80	100
16384	4	80	100
32768	4	80	100
65536	10	80	100

Figure 1: Filemark, Small System, Small-File Workload

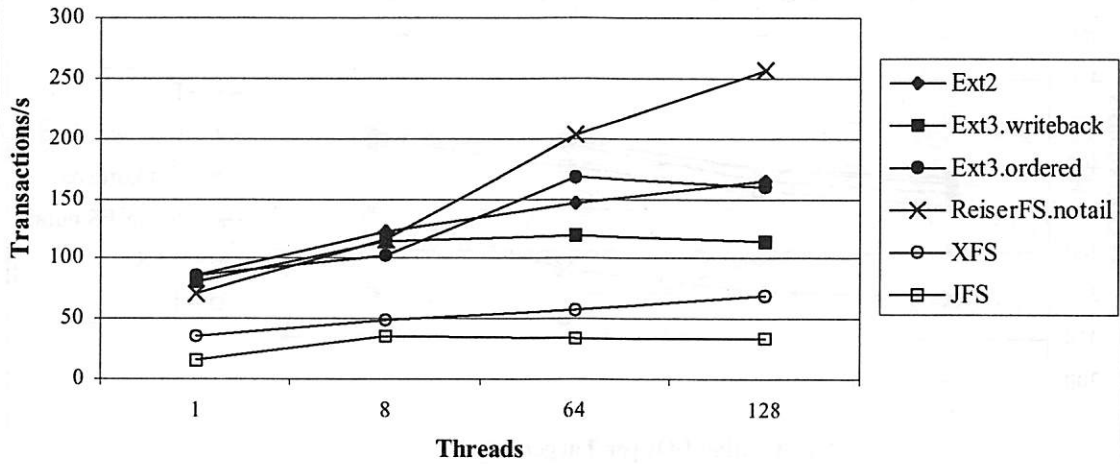


Figure 2: Filemark, Small System, Large-File Workload

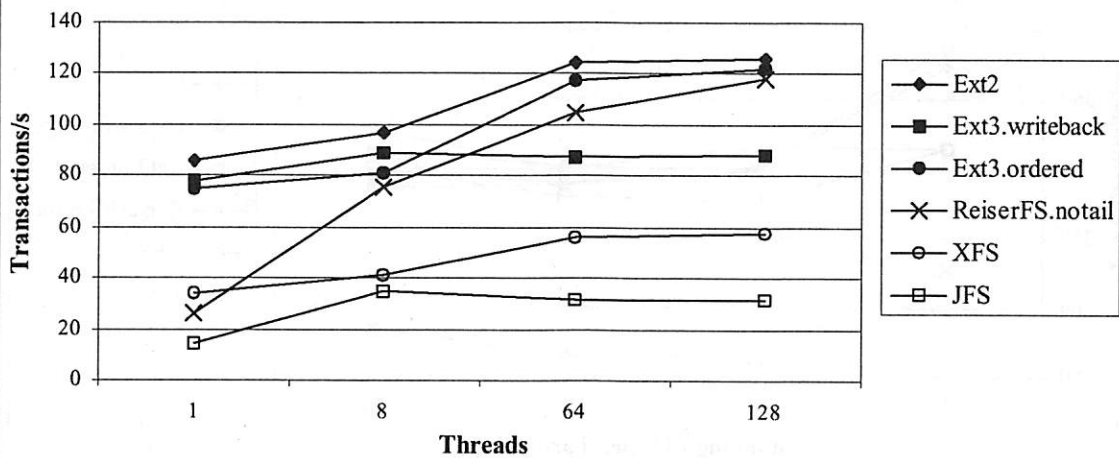


Figure 3: Pgmeter, Medium System, Sequential Read 64 KB Workload

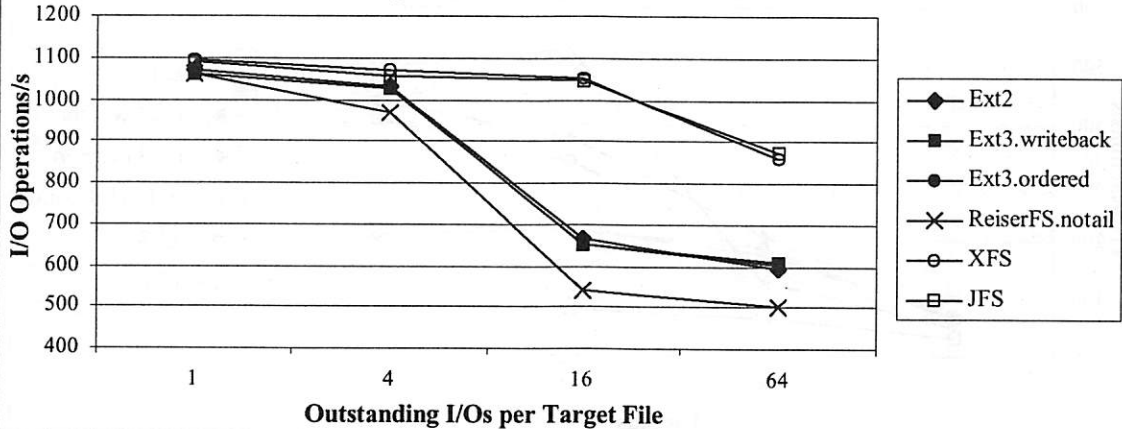


Figure 4: Pgmeter, Medium System, FileServer Workload

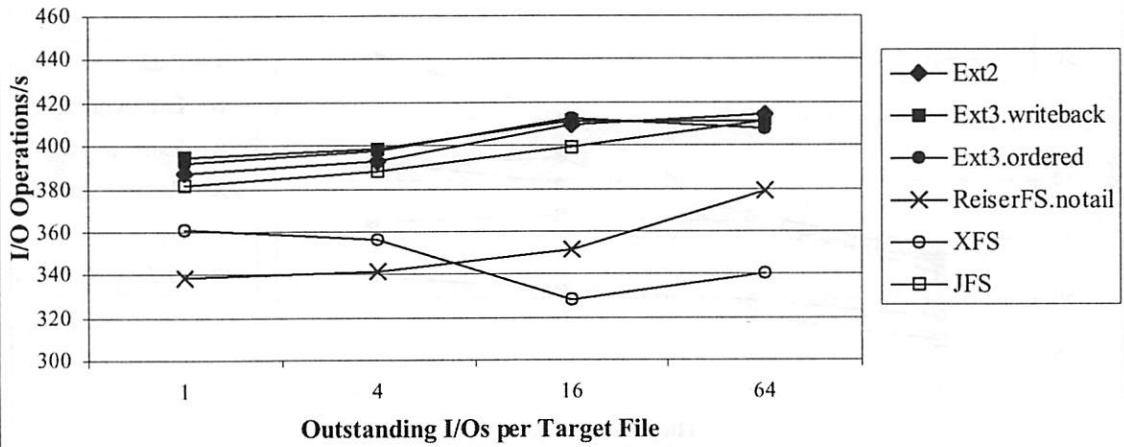


Figure 5: Pgmeter, Medium System, 8K OLTP Workload

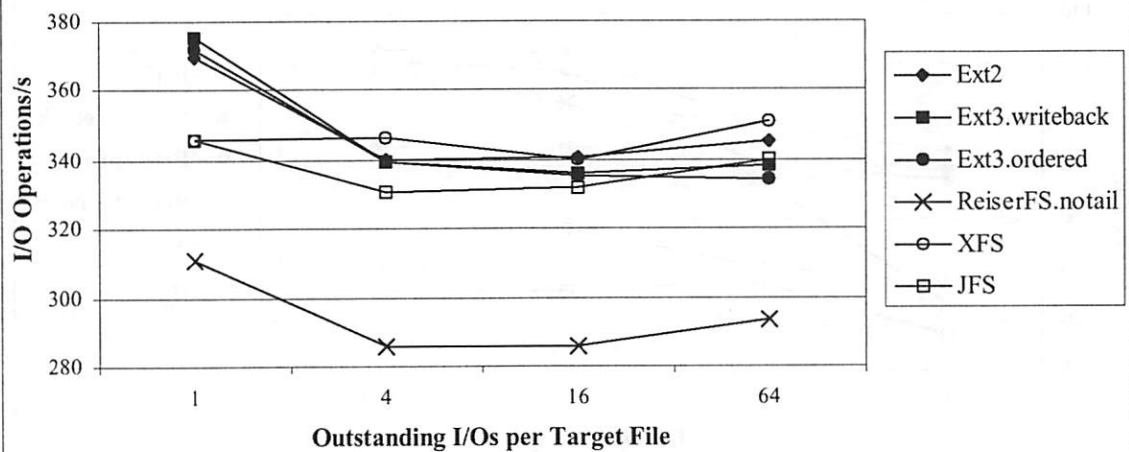
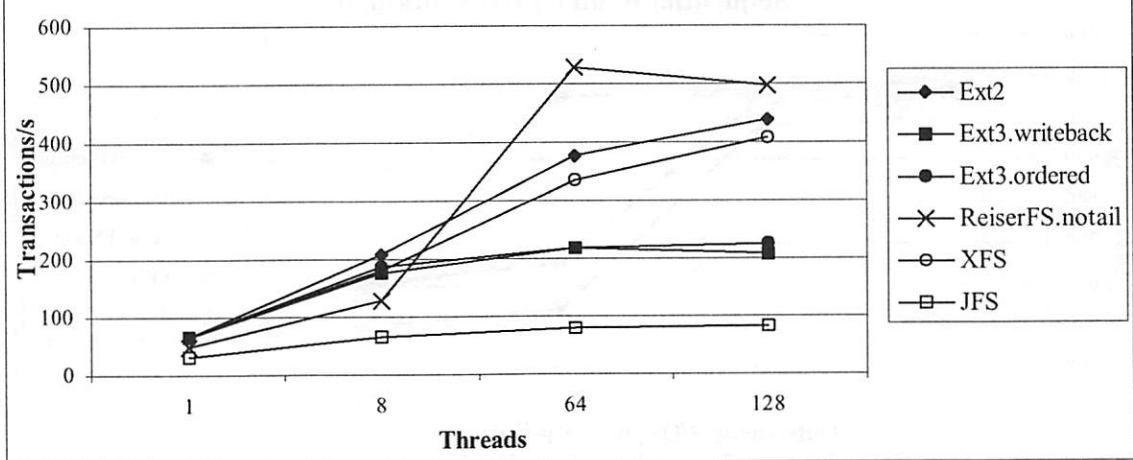
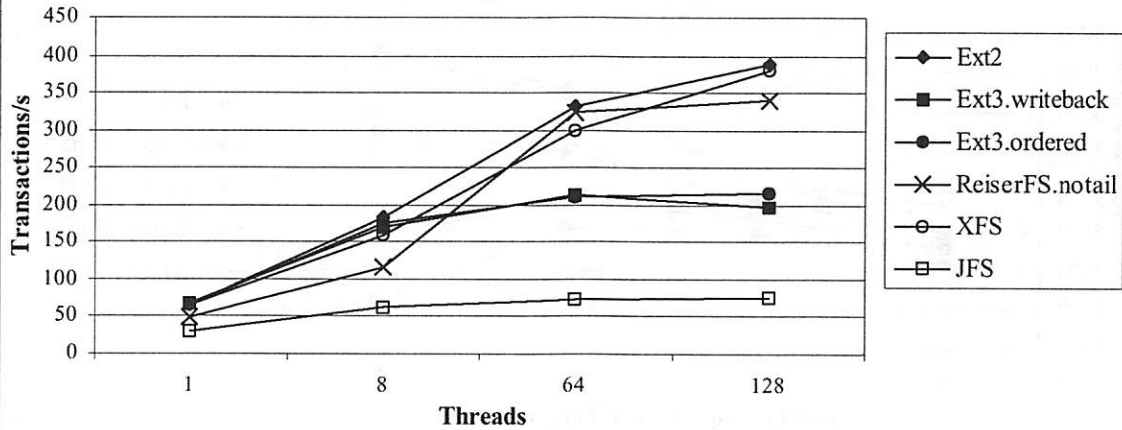


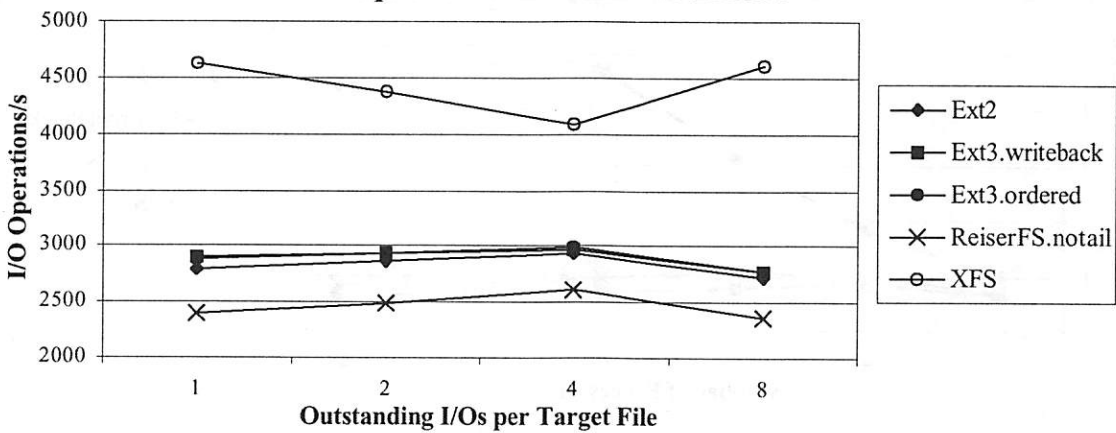
Figure 6: Filemark, Medium System, Small-File Workload



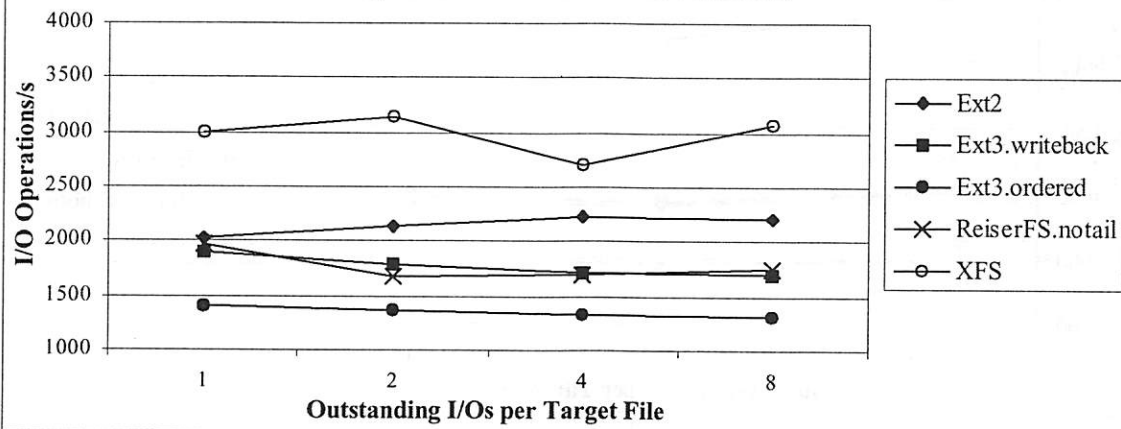
**Figure 7: Filemark, Medium System,
Large-File Workload**

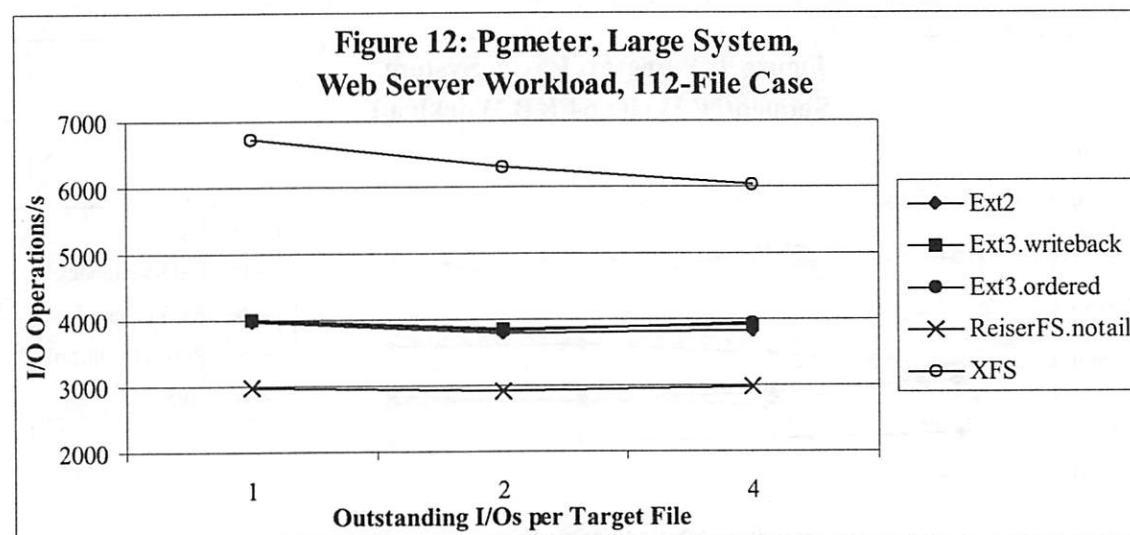
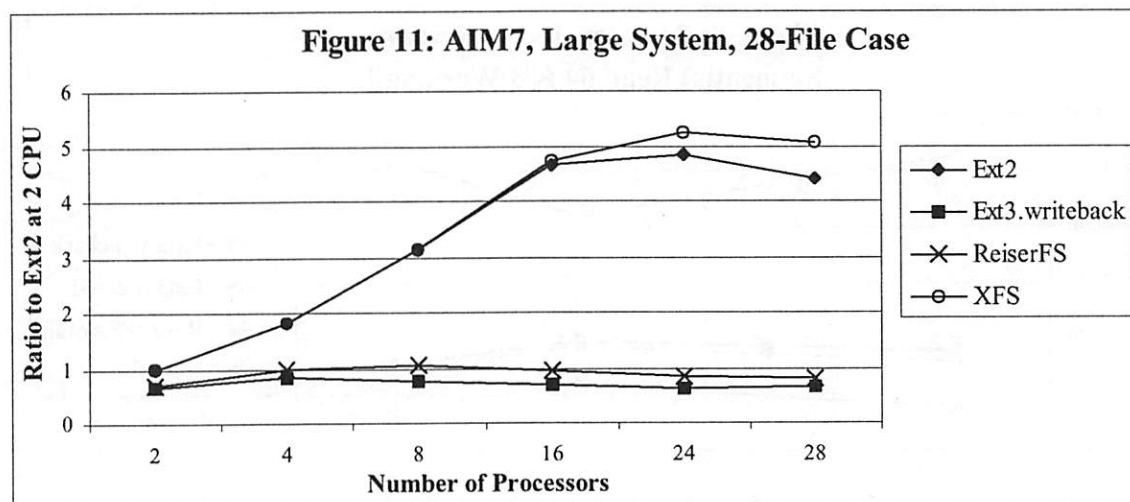
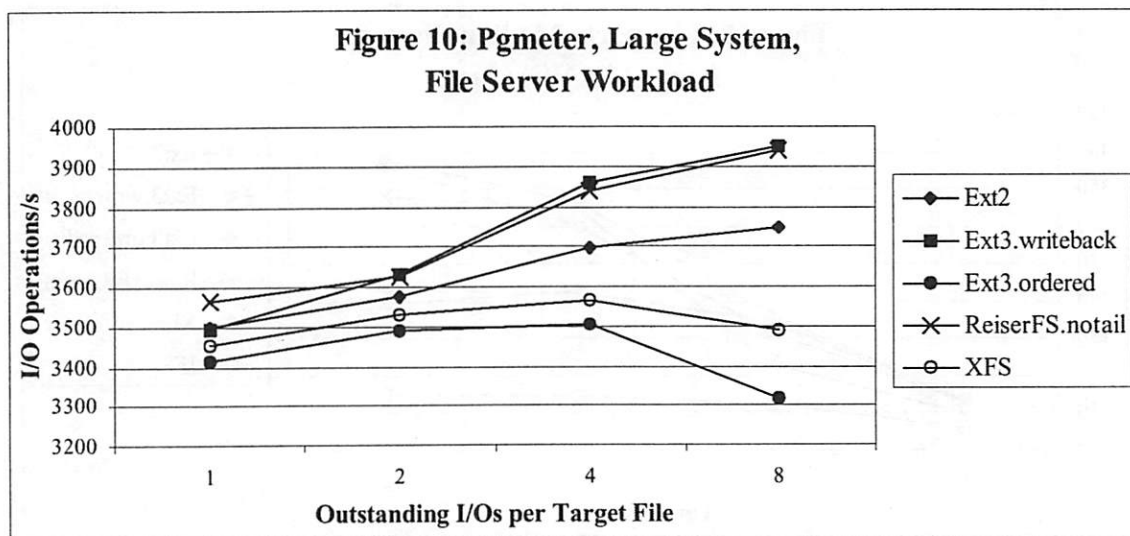


**Figure 8: Pgmeter, Large System,
Sequential Read 64 KB Workload**



**Figure 9: Pgmeter, Large System,
Sequential Write 64 KB Workload**





Speeding Up Kernel Scheduler by Reducing Cache Misses

- Effects of cache coloring for a task structure -

Shuji YAMAMURA, Akira HIRAI, Mitsuru SATO,

Masao YAMAMOTO, Akira NARUSE, and Kouichi KUMON

Fujitsu Laboratories LTD

4-1-1, Kami-Kodanaka, Nakahara-ku, Kawasaki-shi, Kanagawa-pref, Japan 211-8588

{yamamura, ahirai, msato, masao, naruse, kumon}@flab.fujitsu.co.jp

Abstract

In this paper, we propose and evaluate the experimental implementation of cache coloring for task structures on the Linux kernel 2.4.x. We analyzed the behavior of the scheduler from the viewpoint of memory architecture and found that severe cache conflicts occur in a specific cache line. To solve this issue, we applied cache coloring scheme to task structures. Until now, task structures in Linux kernel could not be moved freely in main memory. We noticed, however, that a small modification can enable the coloring. Under heavy workloads, this technique can dramatically reduce cache misses while traversing the run queue that contains a lot of runnable processes. For the evaluation, we used 4-way and 8-way IA server machines. Web server execution and Chat micro benchmark of scheduler intensive benchmarks were used for the measurement. The experimental results showed that the Web server performance achieves a maximum of 23.3% improvement compared to the standard kernel. In the Chat micro benchmark, the message throughput improves a maximum of 89.6% compared to the standard kernel. Furthermore, our coloring technique gives better scalability as the number of processors increases on a SMP system since the lock contention which protects the run queue is reduced.

1 Introduction

Linux has been widely used for enterprise applications, such as web servers and DBMS. In these environments, Linux is expected to be stable and scalable on SMP systems. However, the current Linux (2.4.x) scheduler design contains a well-known performance problem: a single global run queue protected a spin lock. The scheduler

has to traverse the entire run queue to select an appropriate process to run. Therefore as the number of runnable processes increases, traversal time becomes longer. On a SMP system, long traversal time causes both the lock hold time and the lock contention to increase. This limits the scalability of the Linux system. A lot of efforts have been made to improve the scheduler performance by LSE (Linux Scalability Effort) [1]. But no effort focusing on cache and memory architecture has been made yet.

In this study, we observed and analyzed the scheduler behavior from a memory architectural viewpoint using a hardware system bus monitor on the IA32 platform. And we found that a large number of cache misses occur during the run queue traversal. To reduce these cache misses and realize faster traversing, we introduce a new solution into the Linux kernel 2.4.x: *cache coloring for a task structure*.

In this paper, we propose an experimental implementation for coloring which provides large performance gains under heavy workloads. This scheme can reduce cache misses during the run queue traversal and speed up the process scheduling. Furthermore, this method can reduce both the run queue lock hold time and the lock contention on a SMP system.

The rest of this paper is organized as follows: Section 2 summarizes the current scheduler issue in terms of cache efficiency by using a hardware memory bus tracer. Section 3 describes our implementation of cache coloring for a task structure. Section 4 evaluates the performance of our implementation and gives detailed analysis of bus transaction statistics. Section 5 presents the scalability improvement compared with the standard kernel. Section 6 describes related work, and finally we draw conclusions in Section 7.

2 Current Scheduler Issue

The current Linux scheduler has the following two characteristics.

First is a single run queue which is protected by a run queue lock. The run queue is organized as a single double-linked list for all runnable tasks in the system. The scheduler traverses the entire run queue to locate the most deserving process, while holding the run queue lock to ensure exclusive access. As the number of processors increases, the lock contention increases.

Second is the alignment of task structures in physical memory space. The task structure, which contains the process information, is always aligned on an 8KB boundary in physical address space. The problem of this implementation is that each variable in the task structure is always stored at the same offset address of a page frame. This leads to a strong possibility of cache line conflicts for each variable while the scheduler is traversing the long single run queue. At the same time, a large number of cache misses occur.

We observed the memory bus transactions by using hardware bus tracer *GATES* [2]. *GATES* (General purpose memory Access Trace System) can capture memory transactions on the memory bus of a shared memory multiprocessor system during program execution. Figure 1 shows the statistics of memory traffic on a shared memory bus captured by *GATES*. In this observation, 256 apache web server processes (*httpd*) are running on an 4-way Pentium Pro 200 MHz server with 512KB L2-cache each. The vertical axis and the horizontal axis represent the number of transactions per single web request and the

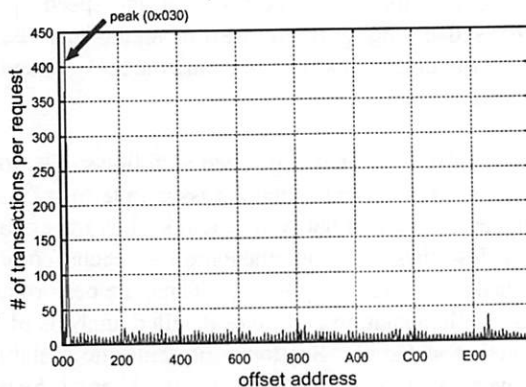


Figure 1: Memory access statistics of running apache on the 4-way Pentium Pro server

page offset address, respectively. As shown in this figure, we can see that a tremendous number of transactions occur on the offset address 0x30. This cache line contains the useful variables of a task structure for scheduling. Figure 2 shows scheduler related variables in a task structure of the 2.4.4 kernel. The 32 bytes block surrounded by the thick line contains the variables which is referred to calculate the priority of each process. And this block also contains the pointer (*run_list.next*) to the next task structure. The scheduler traverses all of task structures on the run queue by referring this pointer. During the run queue traversal, the block of each task structure is successively transferred to the few cache lines corresponding to the page offset address from 0x20 to 0x3f. This causes a lot of cache conflicts on these cache lines, and a large amount of bus transactions are generated seen as Figure 1.

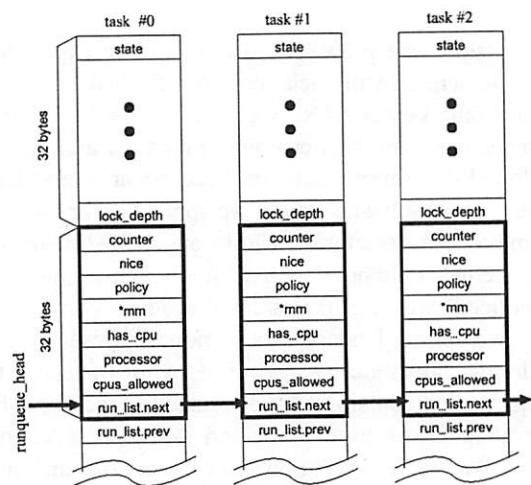


Figure 2: The task structure and the run queue structure

Figure 3 shows the number of transactions to the offset address 0x30 and the web transaction performance. The horizontal axis represents the number of *httpd* processes linked to the run queue. As the number of *httpd* processes on the run queue increases, the number of transactions significantly increases and the web transaction performance correspondingly decreases.

On SMP systems, the run queue is protected by the run queue lock. When traversing the run queue causes frequent cache misses, the time the processor must spend waiting to fill the cache just adds to the overall traversal time. As the traversal time becomes longer, so does the hold time for the run queue lock. On a multiprocessor system, this contention for cache lines translates into increased lock contention, which becomes a bottleneck for scaling.

```
static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__ ("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
    return current;
}
```

(a) original `get_current()`

```
static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__ ("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
    (unsigned long)current |= ((unsigned long)current >> 10) & 0x00000060;
    return current;
}
```

(b) modified `get_current()`

Figure 4: modified `get_current()`

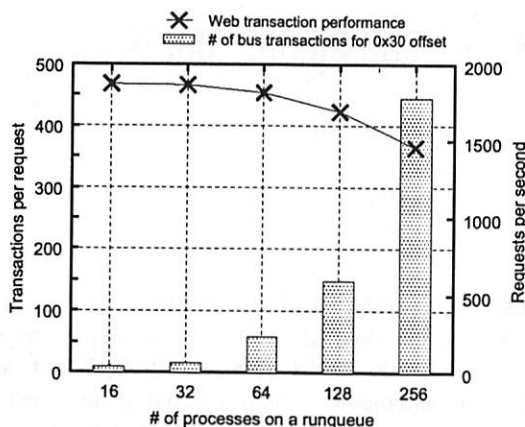


Figure 3: The number of memory bus transactions for the offset address 0x30 and the web server performance

3 Our solution - cache coloring for a task structure -

3.1 Implementation issue for coloring

In general, *cache coloring* [3, 4] is used to address the cache line conflict problem. We expect that the method is able to reduce the cache misses during run queue traversal.

However, cache coloring for the task structure in the Linux kernel seems difficult to implement, because the Linux kernel coding assumes it to be placed on a specific boundary. For example, a procedure to acquire a variable

X in a task structure is as follows:

1. Calling `get_current()` (Figure 4 (a)), the function returns the logical-and value of `%%esp` (current stack pointer) and `0xffffe000`. This is the 8KB boundary of the `%%esp` towards to the address 0.
2. Adding the offset of the requested variable X to the above value.
3. Accessing the process specific variable X using the address.

The kernel stack and task structure share an 8KB allocation of memory aligned on an 8KB boundary. The function `get_current()` assumes that the task structure is always at offset 0 within this block. It zeroes out the low-order bits of the stack pointer to produce a pointer to the task structure.

Because of this implementation, a task structure cannot be shifted freely and the task structure coloring has not been implemented yet.

3.2 Our implementation for coloring

Although the task structure had the above constraints, we noticed that a small modification to the `get_current()` function can enable the coloring.

Figure 4 (b) shows a new `get_current()`. We only inserted the single line in bold face to the original. The

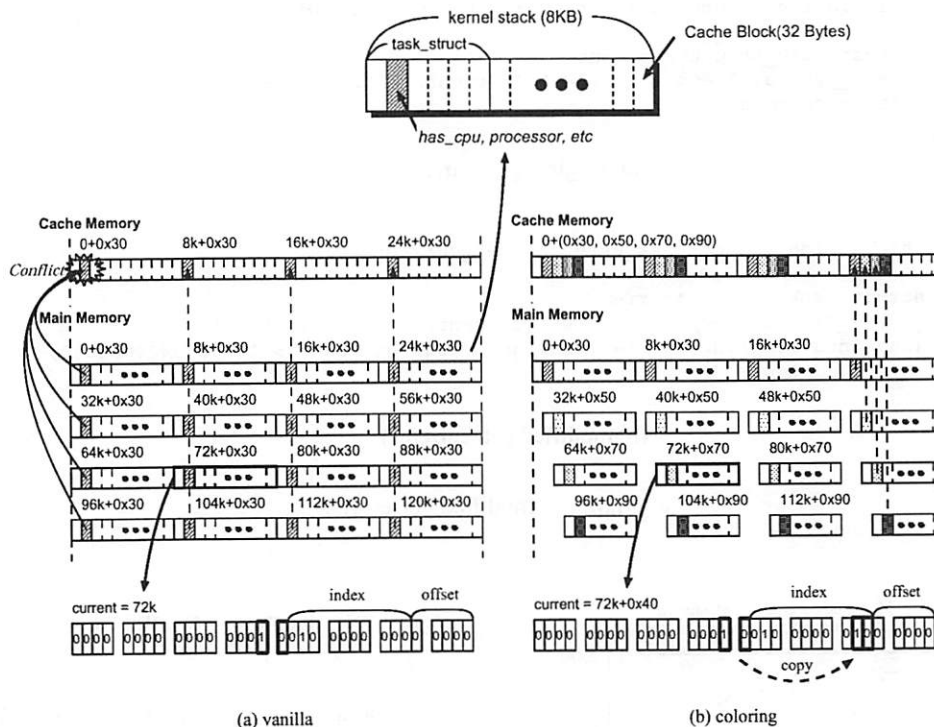


Figure 5: An example of coloring (four colorings)

new `get_current()` returns a slightly different base address, which is shifted in multiples of cache line size. We achieved low overhead by adding only a few bit operations to the original `get_current()`. This is very important because this function is frequently called in the system. Note that this implementation does not require any changes to the core routines of the standard kernel scheduler.

Figure 5 illustrates the implementation of coloring for a task structure. To be easy to understand, we assume that the cache size is 32KB (which probably is smaller than a real L2 cache size) maintained by a direct mapping scheme. The kernel stacks (the task structure objects) shall be contiguously arranged on a main memory (in fact, they are not necessarily arranged continuously in this way, but are aligned on the 8KB boundary on a main memory). The current kernel stack is surrounded by the thick line in this figure.

Figure 5 (a) shows the case of the standard kernel (we call this kernel *vanilla*). The shaded cache blocks are missed frequently. Their memory address is given above each block. As shown in this figure, since each kernel stack is aligned on an 8KB boundary, task structures which are placed at 32KB distance addresses on a main memory are mapped to the same cache lines. If the scheduler con-

tinuously accesses these task structures (such variables as `has_cpu`), cache conflicts occur. For example, grayed data in four kernel stacks of 8KB, 40KB, 72KB, and 104KB in a main memory are transferred to the same line in a cache memory. Therefore, if the scheduler continuously accesses these four task structures, conflicts would occur at the shaded cache lines labeled “8k+0x30”.

On the other hand, Figure 5 (b) shows the case of a kernel using four-coloring. When the coloring kernel calls the modified `get_current()`, it returns 72K+0x40. This is the base address at a 32*2 bytes (equals the size of two cache lines) different offset. In the practical implementation, the modified `get_current()` generates a new current value by hashing it with its upper two bits. In this way, when the scheduler continuously accesses the task structures, cache conflicts never occur.

To apply our implementation to different cache configurations, we have to be careful in our choice of address bits to copy from the masked stack pointer to create the colored pointer to the current task structure. For 2^n colors, we would choose for copying the n bits immediately higher than the bit b satisfying the following equation.

$$b = \log_2 \frac{\text{cache size}}{\text{cache associativity}} \quad (1)$$

Suppose the cache configuration is 512KB with four-way set associative, the address difference of the two blocks on a same line is always multiple of 128KB. This means the lower 17 bits of the two block addresses are always same and using these bits cannot contribute coloring. Thus, to make the cache coloring effective, bits higher than the 17th bit should be used to colorize the structure.

And, we can control the number of colorings by the number of bits to be copied. For example, in order to implement eight colorings, we should use three bits for hashing task structures (replacing “0x60” by “0xe0” in Figure 4 (b)).

The coloring kernel needs modifications in kernel source files other than the one containing `get_current()`. However, the patch file is only 225 lines.

3.3 Negative effect of cache coloring

While the coloring performs effectively so as to reduce cache misses during traversing a run queue, there may be a negative effect on performance. We will explain it with Figure 5.

As shown in Figure 5 (a), many conflicts occur at four cache lines on vanilla kernel. On the other hand, as shown in Figure 5 (b), the number of cache lines used for variables, such as `has_cpu`, are increased to 16. In this case, there is a possibility that the data stored on a colored line before coloring are unfortunately pushed out. As a result, the total amount of bus traffic may increase and lower the overall system performance.

In the latter section for performance evaluation, we will study the effect of our coloring scheme and also analyze such negative influence.

4 Performance Evaluation

The rest of this paper gives quantitative evaluation results of the coloring kernel.

4.1 Methods

To verify the cache coloring improvement, we chose a web server program as a scheduler intensive benchmark,

and evaluated the performance improvement. In the experiment, Apache 1.3.19 is used as the web server, and two Linux kernels are used for comparison: vanilla Linux 2.4.4 kernel and modified kernel with using 32 colors for the task structure. The web clients ran WebBench 3.0 [6] to generate web requests for the server. The specification of the server machine is shown in Table 1. We used three different sizes of L2 cache for evaluating the coloring effects, and all of them have four-way set associative caches. During the benchmark execution, 256 client threads were running on 28 standard PCs and they simultaneously sent requests to the server machine. On the server-side, 256 to a maximum of 1024 server threads (`httpd`)¹ were runnable and were linked on the run queue of the server system.

CPU	Pentium Pro 200MHz × 4
Memory	640MB
L2 cache	256KB, 512KB, 1024KB 4 way set associative
NIC	Intel EtherExpress Pro/100 × 4

Table 1: The specification of the server machine

4.2 Results

At first, we show the performance of the coloring scheme with three different cache configurations, in Figure 6. The horizontal axis is the cache size, and the vertical axis is the performance improvement ratio based on the vanilla kernel. In the experiment, the maximum number of `httpd` processes on a server is set to 256, 512 and 1024, and the number of colors was fixed to 32. We show performance data of the coloring scheme for three different cache configurations.

Second, to measure the coloring effect directly, the number of bus transactions on the memory bus was measured with 1024 `httpd` processes. In the Figure 7, bus transactions are sorted in the following two categories:

colored lines: By coloring, as illustrated in Section 3.2, the run queue list elements are distributed into several cache lines. The total number of bus transactions for these cache lines is expected to decrease due to the coloring.

other lines: The bus transactions of the cache lines other than the above.

¹We changed `MaxClients` parameter value in the configuration file of Apache.

Finally, we measured the number of L2 cache misses and total L2 read counts during the execution of `list_for_each` loop using the performance monitoring counters built into the Pentium processor. This loop traverses the global run queue. Table 2 gives the average of L2 cache miss ratio with three cache sizes. The miss ratio is calculated by the following equation:

$$L2 \text{ cache miss ratio} = \frac{L2 \text{ cache miss counts}}{L2 \text{ read counts}} \quad (2)$$

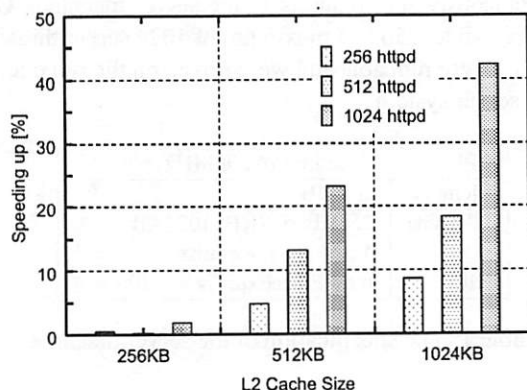


Figure 6: Performance improvement compared to vanilla kernel

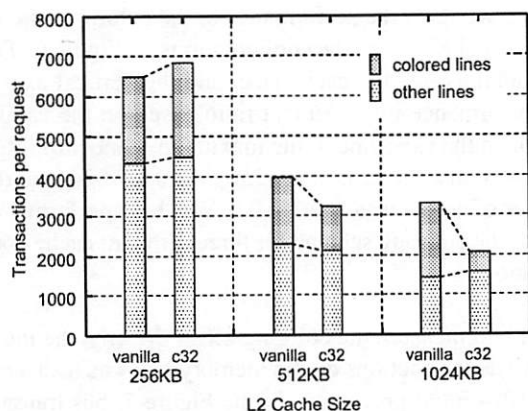


Figure 7: The number of memory bus transactions (1024 httpd processes)

At an L2 cache size of 256KB, we see little or no performance improvement from colorizing (Fig 6). In fact, the number of memory bus transactions per request increased slightly (Fig 7). This is the negative effect of coloring described in Section 3.3. In the case of 32 colorings, $(256KB/8KB) \times 32colors = 1024$ processes ideally should be stored in a cache memory. As shown in Table 2, however, the ratio of L2 cache miss in `list_for_each`

	256 KB	512 KB	1024 KB
vanilla	99.7%	99.6%	99.5%
32 coloring	82.7%	35.8%	8.2%

Table 2: L2 cache miss ratio during searching the run queue (1024 httpd processes)

loop, achieves only 17%-age points reduction. This is because all of task structures are not always contiguously allocated in main memory. Therefore the theoretically possible number of task structures was not stored in cache memory. Since the effect of coloring is partially canceled by the negative effect of increasing bus transactions, there was less performance improvement in the case of 256KB L2 cache.

On the other hand, in Figure 6, significant improvement is observed with both 512KB and 1024KB L2 cache size. The 32 coloring kernel achieves a maximum of 23.1% and 42.3% performance improvement compared to the vanilla one. Also the cache miss ratio is dramatically reduced. Without coloring, the cache miss rate was 99.5% for L2 accesses. Using 32 colors, the L2 cache miss rate decreased to 35.8% and 8.2% with 512KB and 1024KB L2 cache, respectively. As a result, the number of bus transactions for colored cache lines decreases by 33.3% and 73.0% compared to the vanilla kernel (Fig. 7).

As shown above, the cache coloring is more effective for a large cache size, because the large cache area can be utilized to reduce cache conflicts for the run queue.

4.3 The relationship between the number of processes and the number of colorings

In this section, we show the performance difference as the number of colors changes.

By n coloring, the single severely conflicting line in a current scheduler is distributed into n lines. To minimize total cache conflicts, the number of colors should satisfy the following equation:

$$c \geq \frac{p}{(s/8[KB])} \quad (3)$$

In this equation, c , p , and s are the number of colors, the number of processes, and the cache size in KB, respectively. 8KB is the Linux kernel stack size. Ideally, a cache memory can store $s/8$ kernel stacks. For example, in the

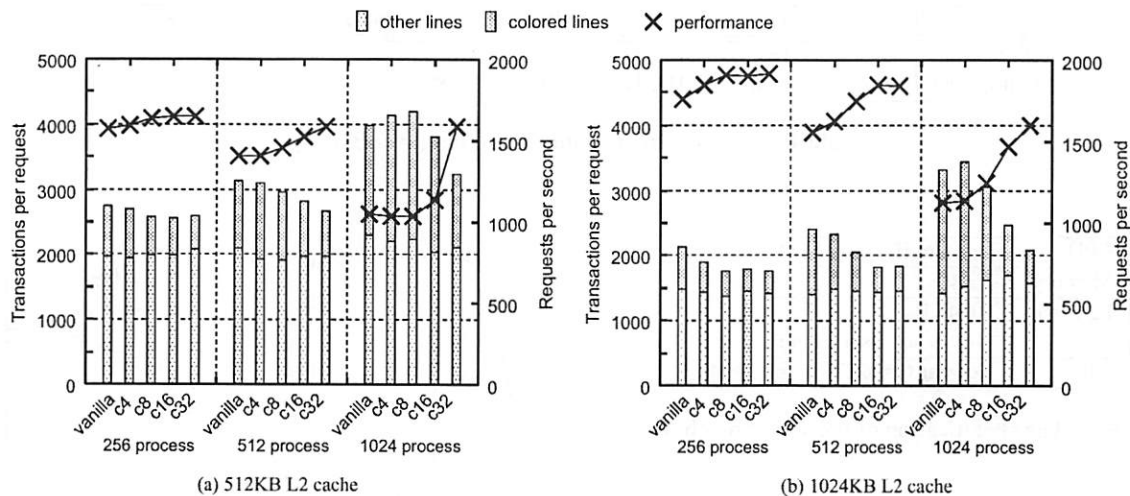


Figure 8: The number of bus transactions and performance changing the number of colorings

case of 1024 processes running on the 512KB cache system, 16 or more colors can prevent conflicts.

To verify the model, we measured the performance varying both the number of colors (4, 8, 16, and 32) and the number of httpd processes (256, 512, and 1024). We used 512KB and 1024KB L2 cache system for the experiment. Figure 8 shows the result of web processing performance and the number of bus transactions. In these graphs, c_n denotes n -coloring kernel.

At first, we discuss the case of 512KB cache size. Using the equation (3), $c = 4, 8$, and 16 colors should be required as a minimum in the case of 256, 512 and 1024 running processes, respectively. As shown in Figure 8 (a), the result almost confirms this requirement: When the number of the coloring is less than the above requirement, the performance is not improved. As the number of coloring increase above the value c of the equation (3), the number of bus transactions for the colored line sharply decreases and the performance improves.

The coloring effect on 1024 KB cache size showed a similar relation between the performance and the number of colorings. 2, 4, and 8 colorings should be used when 256, 512, and 1024 processes are running on a system, respectively. The graph in Figure 8 (b) shows that the number of bus transactions decreases significantly when more than 2, 4, and 8 coloring is used and the total performance improves.

In order for the coloring scheme to be effective, the number of colors must be larger than the threshold selected by equation (3). Otherwise, the effect of the coloring scheme

is hindered by the side-effect of increasing bus transactions with higher cache conflicts.

5 Scalability enhancement with coloring

In this section, we describe the effect of coloring from the viewpoint of scalability. We find that our coloring method is effective in the case of many CPUs.

5.1 Experimental Environment and Benchmarks

The experimental environment is shown in Table 3 and we used the 2.4.4 distribution of the Linux kernel. To evaluate the effect of our coloring method, we ran the following two benchmarks on an 8-way IA server machine: WebBench 3.0 and Chat micro benchmark [7, 8, 9]. A lot of processes are created when these benchmarks are running. In particular, since the run queue length of Chat is longer than that of WebBench, lock contention is expected to be higher. We can expect that the coloring effect in Chat micro benchmark shows larger scalability than in WebBench.

In the case of WebBench, 256 maximum client-threads running on 28 standard PCs, simultaneously send requests to the server machine. During our experimentation, 256 httpd processes are always runnable on the server machine.

	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
32 coloring	8.2%	13.3%	15.8%	17.6%	18.0%	20.6%	22.6%	23.3%
Multi-Queue	-3.0%	13.6%	18.2%	21.7%	23.8%	26.9%	28.7%	30.6%

Table 4: Speeding up to vanilla scheduler (WebBench)

CPU	Pentium III Xeon 550MHz × 8
Memory	1GB
L2 cache	1024KB (4-way set associative)
NIC	Netgear GA622T (1GbE) × 4

Table 3: The specification of the server machine

The Chat micro benchmark is a stand-alone type benchmark, and no client machine is used. This benchmark simulates chat rooms with multiple users exchanging messages using TCP sockets. Each chat room consists of 20 users, and each user broadcasts a number of 100 byte messages in the room. To handle message exchange, one user program creates four threads. Thus 80 threads are created per room. The characteristic parameters of the Chat micro benchmark are the number of rooms and the number of messages per user. We choose 30 rooms and 300 messages per user as parameters, so that 2400 processes (threads) are generated on a system during experimentation.

In this research, we also evaluated the multi-queue [8] (we called MQ) scheduler proposed by IBM, which is an alternative to vanilla scheduler. MQ scheduler separates the run queue and the run queue lock for each CPU in the system.

5.2 Result

5.2.1 WebBench

Figure 9 shows WebBench results for the following three kernel: standard kernel (vanilla), 32 coloring kernel (c32), and multi-queue scheduler kernel (MQ). Table 4 shows performance gains of c32 and MQ compared with vanilla.

We can see in this graph that both of c32 and MQ show better performance than vanilla. In Table 4, c32 and MQ achieves maximum of 23.3% and 30.6% improvement on an 8-way, respectively. In fact, MQ shows more performance improvement than c32. However, MQ requires more extensive changes to be implemented than c32. Although requiring less modification, c32 nearly achieves

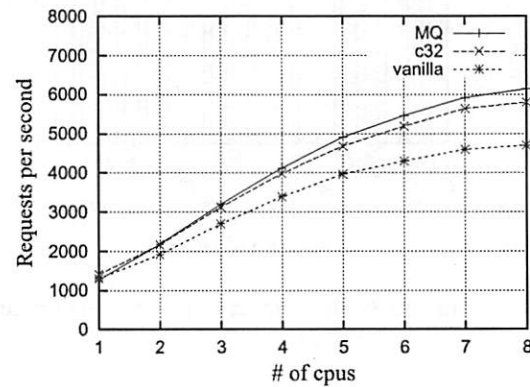


Figure 9: WebBench Performance (# of httpd processes: 256)

the performance improvement of MQ. Furthermore, c32 achieves speedup over all of the CPU set. In contrast, MQ's performance drops on single CPU system.

In Table 4, as the number of CPUs increases, c32 gains larger speeding up. When the number of CPUs is one, the performance gains is 8.2%. When the number of CPUs becomes 8, the performance gains 23.3%. The major reason is the improved lock statistics. The reduction of cache misses in scheduler makes the traversal time shorter and dramatically decreases the holding time for the run queue lock. Thus, our coloring method could perform more effectively on a SMP system rather than a uniprocessor system.

To confirm this, we also measured following two characteristics.

L2 cache miss ratio: measuring the number of L2 cache miss ratio during the execution of `list_for_each` loop. The miss ratio is calculated by the equation (1) as described in Section 4.2. We used performance monitoring counters built into the Pentium processor.

Lock hold time and lock contention: measuring following two statistics: (1) the time that the run queue lock is held, and (2) the fraction of lock requests that found the run queue lock was busy when it was requested. These information are measured by using

	1CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	92.4%	98.0%	98.0%	92.4%	96.2%	94.6%	92.3%	93.9%
32 coloring	6.0%	7.5%	6.8%	7.2%	6.6%	7.4%	11.1%	13.7%

Table 5: L2 cache miss ratio during run queue traversal (WebBench)

		2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	Contention	9.2%	15.4%	19.7%	22.9%	27.2%	35.9%	45.6%
	Hold Mean [us]	42	40	40	41	43	43	36
	Hold Max [us]	133	137	143	157	153	153	156
32 coloring	Contention	2.3%	4.1%	6.0%	7.2%	9.3%	14.0%	23.4%
	Hold Mean [us]	12	12	13	13	13	11	13
	Hold Max [us]	112	129	113	135	132	78	80

Table 6: lock statistics for run queue lock (WebBench)

Lockmeter tool [10].

The results are shown in Table 5 and 6, respectively. In the case of vanilla kernel, the `list_for_each()` loop involves potentially large cache misses as seen in Table 5, resulting more than 90% cache miss ratio. In contrast, 32 coloring shows substantial reduction to about 10% on any number of CPUs system. Moreover, in Table 6, both of the mean of lock hold time and the lock contention are largely reduced on any number of CPUs system by using 32 coloring. The lock contention is reduced from 45.6% to 23.4% on an 8-way system. This leads to better scalability than vanilla kernel.

5.2.2 Chat

Figure 10 shows the throughput performance on the standard kernel (vanilla), 32 coloring kernel (c32), and multi-queue scheduler kernel (MQ).

As expected, c32 shows significantly better throughput than vanilla. Vanilla kernel slightly scales up to 4 CPUs, but its throughput performance starts dropping from 5 CPUs upward. On the contrary, c32 scales up to 6 CPUs which provides 89.6% throughput improvement compared to vanilla. MQ gains the best throughput among these three kernels.

As similar in WebBench, we collected information for L2 cache miss ratio and lock statistics. The results are shown in Table 7 and 8, respectively. We can see that there is a substantial reduction in L2 cache miss ratio on any number of CPUs, leading to speedup of the run queue traversal. This result shows that the lock hold time is reduced

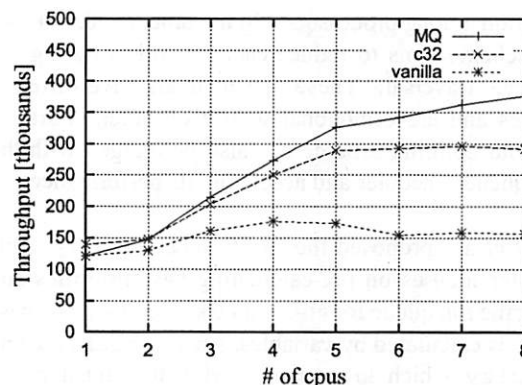


Figure 10: Chat Performance (30 rooms, 300 messages)

from 40us to 14us on the 8 CPUs system by the coloring scheme. The lock contention is also decreased from 85.8% on vanilla to 69.7% on c32. The run queue lock contention of Chat micro benchmark is higher than that of WebBench.

6 Related work

Several schemes for speeding up Linux scheduler have been proposed.

Kravetz et al. proposed the multi-queue scheduler to enhance scalability of the Linux 2.4.x on large scale SMP machines [8]. The multi-queue scheduler separates the global run queue into a number of queues and distributes them to each CPU. Each CPU maintains its own run queue. This scheduler aims to reduce the run queue lock

	1CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	99.7%	99.8%	76.4%	98.8%	64.5%	85.3%	94.4%	84.8%
32 coloring	3.2%	4.6%	2.8%	2.4%	5.8%	3.6%	2.8%	3.3%

Table 7: L2 cache miss ratio during run queue traversal (Chat)

		2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	Contention	33.0%	48.4%	51.9%	73.0%	75.0%	85.2%	85.8%
	Hold Mean [us]	113	51	32	50	34	54	40
	Hold Max [us]	352	196	125	186	147	270	155
32 coloring	Contention	23.6%	38.7%	48.3%	56.4%	60.6%	61.9%	69.7%
	Hold Mean [us]	26	25	18	19	17	13	14
	Hold Max [us]	235	215	192	200	168	133	166

Table 8: lock statistics for run queue lock (Chat)

contention among processors. On the other hand, our coloring scheme aims to reduce cache conflicts during the run queue traversal. These two methods have different purposes and are orthogonal with each other. Furthermore, our coloring scheme can also be merged with the multi-queue scheduler and accelerate its performance.

Molloy et al. proposed the ELSC scheduler [11]. This scheduler focuses on pre-calculating base priorities and sorting the run queue for efficient task selection. The base priority is calculated by variables, such as `counter` and `priority`, which do not change while the runnable task is not running on a processor. This scheduler maintains a table which is sorted in the base priority. Each entry of the table contains a double-linked list which involves processes with same priority. The Priority Level Scheduler (PLS) [8] is proposed as a similar scheme. Both ELSC and PLS can make scheduling decisions faster, but cannot eliminate run queue lock contention, therefore do not show further scalability as the number of CPUs increases.

In [12], Sears has pointed out severe cache misses, which occur in the current scheduler, and provided the solution by using a prefetch technique. This improvement has already been merged to the latest kernel (after 2.4.10 version). However, in order for the prefetch technique to perform efficiently, the memory access latency and `goodness()` execution must overlap almost perfectly. The potential problem of this method is that these values depends on the memory system configuration.

Development of the Linux kernel is performed very rapidly on the Linux kernel mailing list. We have already contributed our scheme as a patch. Concurrently, Spraul has also contributed a patch with another implementation for coloring task structures. In his implementation, the

kernel allocates task structures through the Slab Allocator [4, 13]. His implementation does not fix the number of colors nor does not depend on the cache configuration. In this respect, it is much more general than ours. On the other hand, our approach needs only a small modification to `get_current()` function, and the coloring effect could be quickly verified. Using our implementation, we can control the number of colors. Therefore, we could analyze the effect of the coloring, reduction of bus traffic, L2 cache misses and lock contentions, as a function of the number of colors.

7 Conclusions

We showed in this paper that the current Linux kernel potentially has scalability problem due to severe cache line conflicts from the placement of task structures in physical memory. We observed memory bus transactions on real SMP server systems and confirmed that large number of cache misses occur in the scheduler under heavy workload.

To address this issue, we proposed and implemented the cache coloring for a task structure. The evaluation result of this implementation demonstrates that the cache miss ratio while traversing the run queue is significantly reduced and the scheduling speed is enhanced. In WebBench, the web transaction performance on the coloring kernel achieved maximum of 42.3% improvement on 4-way Pentium Pro 200MHz system and 23.3% improvement on 8-way Pentium III Xeon 550MHz system. In Chat benchmark, the message throughput on the coloring kernel showed a maximum of 89.6% improvement on 8-way Pentium III system.

Reduction of cache misses can lead to decreasing run queue traversal time. On a SMP system this results in decreasing the lock hold time and lock contention. This is the effect of coloring on a large scale SMP machine. We verified these effects on an 8-way system, and found that the coloring scheme achieves better scalability than the standard kernel.

On the other hand, there is potential disadvantage caused by coloring: useful data are replaced on colored lines. To avoid this problem, we provided a simple model to decide the appropriate number of colorings, and verified the model with the bus transactions data observed on a real system.

As the gap between processor and memory speed grows wider the cache conflict issue caused by the current scheduler becomes more serious. Our coloring scheme is an essential technique for ameliorating this issue. The coloring scheme patch is contributed to the open source community, and is freely available for use and modification. The current patch can be downloaded from <http://www.labs.fujitsu.com/en/techinfo/linux/>.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. We would like to thank Kazuhiro Matsumoto and Andreas Savva for their helpful comments. In addition, we would like to thank the people on the Linux kernel mailing list who provided us with valuable comments.

References

- [1] "Linux Scalability Effort Project", <http://sourceforge.net/projects/lse>
- [2] "Development of GATES (Memory Access Trace System for a PC Server)", Mitsuru SATO, Akira NARUSE, Kouichi KUMON, Proceedings of the 59th National Convention IPSJ, September 1999 (in Japanese).
- [3] "Solaris Internals Core Kernel Architecture", Jim Mauro and Richard McDougall, SUN MICROSYSTEMS PRESS.
- [4] "The Slab Allocator: An Object-Caching Kernel Memory Allocator", Jeff Bonwick, In USENIX Conference Proceedings, pp 87-98, 1994.
- [5] "Understanding the Linux Kernel", Daniel P. Bovet, Marco Cesati, O'Reilly & Associates, pp 69-70, October 2000.
- [6] "WebBench Homepage", <http://etestinglabs.com/benchmarks/webbench/webbench.asp>
- [7] "Java Technology, Threads, and Scheduling in Linux", R. Bryant and B. Hartner, Java Technology Update, 4(1), Jan 2000.
- [8] "Enhancing Linux Scheduler Scalability", Mike Kravetz, Hubertus Franke, Shailabh Nagar, Rajan Ravindran, 5th Annual Linux Showcase & Conference, November, 2001.
- [9] "Linux Benchmark Suite Homepage", <http://lbs.sourceforge.net/>
- [10] "Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel", Ray Bryant, John Hawkes, 4th Annual Atlanta Linux Showcase & Conference, October, 2000.
- [11] "Scalable Linux Scheduling", S.Molloy and P. Honeyman, In CITI Technical Report 01-7, University of Michigan, May 2001.
- [12] "The Elements of Cache Programming Style", Chris B. Sears, 4th Annual Atlanta Linux Showcase & Conference, October, 2000.
- [13] "UNIX Internals: The New Frontiers", Uresh Vahalia, Prentice Hall.

Overhauling Amd for the '00s: A Case Study of GNU Autotools

Erez Zadok
Stony Brook University
ezk@cs.sunysb.edu

Abstract

The GNU automatic software configuration tools, Autoconf, Automake, and Libtool, were designed to help the portability of software to multiple platforms. Such *autotools* also help improve the readability of code and speed up the development cycle of software packages. In this paper we quantify how helpful such autotools are to the open-source software development process. We study several large packages that use these autotools and measure the complexity of their code. We show that total code size is not an accurate measure of code complexity for portability; two better metrics are the distribution of CPP conditionals in that code and the number of new special-purpose Autoconf macros that are written for the package.

We studied one package in detail—Am-utils, the Berkeley Automounter. As maintainers and developers of this package, we tracked its evolution over ten years. This package was ported to dozens of different platforms and in 1997 was converted to use GNU autotools. We show how this conversion (autotooling) resulted in a dramatic reduction in code size by over 33%. In addition, the conversion helped speed code development of the Am-utils package by allowing new features and ports to be integrated easily: for the first year after the conversion to GNU autotools, the Am-utils package grew by over 70% in size, adding many new features, and all without increasing the average code complexity.

1 Introduction

Large software packages, especially open-source (OSS) ones, must be highly portable so as to maximize their use on as many systems as possible. Past techniques for ensuring that software can build cleanly and run identically on many systems include the following:

- Asking users to manually configure a package prior to compilation by editing a header file to turn on or off package features or to specify services available from the platform on which the package will be run. This process required intimate knowledge of the system on which the package (e.g., C-News) was to be built.
- Trying to achieve portability using CPP macros and nested `#ifdef` statements. Such code results in complex, system-specific, deeply-nested CPP macros which are hard to maintain.
- Using Imake [2], a system designed specifically for building X11 applications. Imake defines frozen configurations for various systems. However, such static configurations cannot account for local changes made by administrators.
- Using Metaconfig [11], developed primarily for building Perl. This system executes simple tests similar to Autoconf, but users are often asked to confirm the results of these tests or to set the results to the proper values. Metaconfig requires too much user interaction to select or confirm detected features and it cannot be extended as easily as Autoconf.

GNU Autoconf [5] solves the above problems by providing canned tests that can dynamically detect various features of the system on which the tests are run. By actually testing a feature before using it, Autoconf and its sister tools Automake [6] and Libtool [7] can build packages portably without user intervention. These automated software configuration tools (*autotools* [10]) can run on numerous systems. Autotools work identically regardless of the OS version, any local changes that administrators installed on the system, which system software packages were installed or not, and which system patches were installed.

The rest of this paper is organized as follows. Section 2 explains the motivation for automated software configuration tools. In section 3 we explain how GNU Autoconf and associated tools work, explore their limitations, and describe how we used these tools. Section 4 evaluates several large OSS packages and details the use of autotools in the Am-utils package. We conclude in Section 5.

2 Motivation

When software packages grow large and are required to work on multiple platforms, they become more difficult to maintain without automation. We spent several years maintaining Amd and Am-utils, as well as fixing, porting, and developing other packages. During that time we noticed how difficult it was to maintain and port such packages and that led us to convert Amd to use autotools. As a result of the conversion, we noticed that Am-utils became easier to maintain and port. We therefore set out to quantify this improvement in the portability and maintainability of the Am-utils package, and those investigations led to writing this paper.

There are six reasons why porting such packages to new platforms, adding new features, or fixing bugs becomes a difficult task more suitable for automatic configuration:

1. **Operating system variability:** There are more Unix systems available today, with more minor releases, and with more patches. Flexible software packaging allows administrators to install selective parts of the system, increasing variability. An automated build process can track small changes automatically, and can even account for local changes.
2. **Code inclusion and exclusion:** To handle platform-specific features, large portions of code are often surrounded by `#ifdef` directives. Platform-specific code is mixed with more generic code. Often, system-specific source files are compiled on every system, because there is no automatic way to compile them conditionally.
3. **Multi-level nested macros:** To detect certain features reliably, older code uses deeply nested `#ifdef` directives. This results in complex macro expressions designed to determine features as reliably as possible. The main problem with such macros is that they provide second-hand or anecdotal knowledge of the system. For example, to test if a compiler supports "void *", some code depends on the name of the compiler (GNUC) rather than directly testing for that feature's existence.
4. **Shared libraries:** Many packages need to build and use shared or static libraries. Such packages often support shared libraries only on a few systems (e.g., Tcl before it was autotooled), because of differing shared library implementations. Frequent use of non-shared (static) libraries results in duplicated code that wastes disk space and memory.
5. **Human errors:** Manually-configured software is more prone to human errors. For example, the first port of Amd to Solaris on the IA32 platforms copied the static configuration file from the SPARC platform, incorrectly setting the endianness to big-endian instead of little-endian.
6. **Novice and overworked administrators:** With a rapidly growing user base and the growth of the Internet, the average expertise of system administrators has decreased. Overworked administrators cannot afford to maintain and configure many packages manually.

Converting OSS packages to use GNU autotools—Autoconf [5], Automake [6], and Libtool [7]—addresses the aforementioned problems in five ways:

1. **Standard tests:** Autoconf has a large set of standard portable tests that were developed from practical experiences of the maintainers of several GNU packages. Autoconf tests for features by actually exercising those features (e.g., compiling and running programs that use those features). Packages that use Autoconf tests are automatically portable to all of the platforms on which these tests work.
2. **Consistent names:** Autoconf produces uniform macro names that are based on features. For example, code which uses Autoconf can test if the system supports a reliable `memcmp` function using `#ifdef HAS_MEMCMP`, rather than depending on system-specific macros (e.g., `#ifndef SUNOS4`). Autoconf provides a single macro per feature, reducing the need for complex or nested macro expressions. This improves code readability and maintainability.
3. **Shared libraries:** By using Libtool and Automake along with Autoconf, a package can build shared or static libraries easily, removing a lot of custom code from sources and makefiles.
4. **Human factors:** Building packages that use autotools is easy. Administrators are becoming increasingly familiar with the process and the standard set of features autotools provide (i.e., run `./configure` and then `make`). Administrators do not need to configure the software package manually prior to

compilation and they are likely to make fewer mistakes. This standardization speeds up installation and configuration of software.

5. **Extensibility:** Finally, software maintainers can extend Autoconf by writing more tests for specific needs. For example, we wrote specific tests for the Am-utils package that detect its interaction with certain kernels. This allowed us to separate the common code from the more difficult-to-maintain platform-specific code.

Our experiences with maintaining the Amd package clearly show the benefits of autotools. When we converted the Amd package [9, 12] to use autotools, the code size was reduced by more than one-third and the code became clearer and easier to maintain. Fixing bugs and adding new features became easier and faster, even major features that affected significant portions of the code: NFSv3 [8] support, Autofs [1] support, and a run-time automounter configuration file `/etc/amd.conf`. New features that we added immediately worked on many supported systems and bugs fixes did not introduce additional bugs.

3 Autotooling

The basic idea behind Autoconf is to determine a feature's availability by running a small test that actually uses the feature. The test often writes a small program on the fly, compiles it, and possibly links and runs it. This is a reliable method of detecting features. Autoconf-generated configuration scripts are portable. When run, they use portable shell scripting and tools such as `sed` and `grep`. Building `configure` scripts requires GNU M4, but only by the maintainers of the package, not by those building the package.

In this section we explain how GNU autotools work and detail the types of autotool tests we used or wrote for use with Am-utils.

3.1 Autoconf

An Autoconf `configure` script is built from a `configure.in` file that contains a set of Autoconf M4 macros to use. Autoconf translates the M4 macros into their respective portable shell code. For example, to test if the system supports the `bzero` function, we used this M4 macro: `AC_CHECK_FUNCS(bzero)`. Autoconf translates this M4 macro call into shell code that creates, compiles, and links the following program:

```
#include "confdefs.h"
char bzero(); /* forward definition */
int main() {
    bzero();
    return 0;
}
```

If the program compiles and links successfully, the `configure` script defines a CPP macro named `HAVE_BZERO` in an automatically-generated configuration file named `config.h`. This macro is based on the existence of the feature and can be used reliably in the sources for the package. Note that it is not necessary to run this program to determine if `bzero` exists. In fact, the above program will fail to run properly because the `bzero` function was not given proper parameters.

Assuming the package also checks for the existence of the `memset` function, the maintainers of the package can use the following portable code snippet reliably:

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif /* HAVE_CONFIG_H */

#ifdef HAVE_BZERO
# ifdef HAVE_MEMSET
#   define bzero(ptr, len) \
        memset((ptr), 0, (len))
# else /* not HAVE_MEMSET */
#   error neither bzero nor memset found
# endif /* not HAVE_MEMSET */
#endif /* not HAVE_BZERO */

...
struct nfs_args na;
bzero(&na, sizeof(struct nfs_args));
```

There are four categories of feature tests that we used or developed during the autotooling process for Am-utils:

1. No Autoconf test needed.
2. Simple existing Autoconf tests were available.
3. New simple Autoconf tests had to be written.
4. Static Autoconf tests had to be written.

3.1.1 No Test Needed

These features are common to most Unix platforms and they are easily available from standard system header files. No Autoconf tests or macros were required; the user need only include the correct header files. For example, to detach from its controlling terminal, one of the methods a long-running daemon such as `amd` uses is the `TIOCNOTTY` ioctl. By simply including the right header files, we could write C code that performed an action only if `TIOCNOTTY` was defined.

3.1.2 Simple Existing Tests

These are Autoconf macros for which an existing test was suitable. Over 70% of the Autoconf tests used by Am-utils and other large packages fall into this category. A few examples are:

`AC_CHECK_LIB(rpcsvc, xdr_fhandle)` checks if the `rpcsvc` library includes the `xdr_fhandle` function. If so, the test ensures that `-lrpcsvc` is used when linking binaries.

`AC_REPLACE_FUNCS(strdup)` tests if the `strdup` function exists in any standard library, and if not, adds `strdup.o` to the objects to build. The package maintainers are then required to supply a replacement `strdup.c` file.

`AC_CHECK_HEADERS(nfs/proto.h)` checks if that header file exists. If so, the test defines the macro `HAVE_NFS_PROTO_H`, which can be used in the code to include the header file only if it exists.

`AC_CHECK_TYPE(time_t, unsigned long)` looks for a definition of `time_t` in standard header files such as `<sys/types.h>`. If not found, the macro defines the type to `unsigned long`. Portable code need only use `time_t`.

`AC_FUNC_MEMCMP` tests if the `memcmp` library function exists and if it is 8-bit clean; some versions of this function incorrectly compare 8-bit data. This is an example of how Autoconf can help to detect bugs in system software and offer the package maintainer a workaround option.

Interestingly, the full set of all Autoconf tests also serves as a testament to the total sum of operating system variability. The macros list numerous features that possibly differ from system to system.

3.1.3 Newly Written Tests

These were macros for which no existing Autoconf test existed and thus had to be written. The key here is to design and write tests that could *reliably* determine a feature all of the time, regardless of the tools used.

We describe only one example in this paper: `AC_CHECK_FIELD`, a test to determine if a given structure contains a certain field.¹ This test can be used to handle structures with the same name that have different members across different platforms. Our M4 test macro is used as follows:

```
AC_CHECK_FIELD(struct sockaddr, sa_len)
```

The macro takes two arguments: the first (\$1) is the name of the structure and the second (\$2) is the name

of the field. The macro creates and tries to compile the following small program:

```
main() {
    $1 a;
    char *cp = (char *) &(a.$2);
}
```

If the above program compiles successfully, the test defines a CPP symbol whose name is automatically constructed: `HAVE_FIELD_STRUCT_SOCKADDR_SA_LEN`. Code that uses this CPP symbol can perform the proper actions when the `sockaddr` structure contains an `sa_len` field.

3.1.4 Static Autoconf Tests

Autoconf tests must be 100% deterministic. We identified several classes of problems where a simple reliable test could not be written. We wrote these tests statically: the result of the test is hard-coded based on the operating system name and version. These include tests for certain types, for kernel features, for system bugs, and for other features which would have required complex tests.

1. **Types:** The types of arguments passed to functions or used in structure fields cannot be detected easily. C is not a type-safe language. Some C compilers do not always fail when a type mismatch occurs, even with special compiler flags. One example of such a test is determining the type of the NFS file handle field in `struct nfs_args`: it can be a fixed-size buffer, one of several other structures, or a pointer to any of those.
2. **Kernel features:** Kernel internals are difficult to probe, even with root access. For example, determining how the `mount(2)` system call works is not practical because it requires mounting a known file system and superuser privileges. Worse, if the resource being mounted is a remote file system (e.g., NFS), the client also needs to know the name of the server exporting that file system and the name of the exported path on the server.
3. **System bugs:** Some systems have bugs that cannot be easily detected. For example, all versions of Irix up to 6.4 use an NIS function `yp_all` that leaks a TCP file descriptor opened to the bound NIS server `ypserv`. To detect this bug, a program must run on a configured NIS system and know which NIS maps to download—information that is specific to the site. We considered and rejected several more complex alternatives to detect this bug; it was simpler to hard-code the answer.

4. **Complex tests:** If the Autoconf test being written is too complex or long, generally more than half a page of M4, C, and sh code, and is used only once or twice throughout the configure script, it is better to write it statically. A shorter test, even a static one, is often more readable and helps to reduce the maintenance effort needed for the package.

The following example illustrates when a static test was preferable. Automounters include an entry in mount tables (e.g., /etc/mtab) that is set as “hidden” from the `df(1)` program because there is not much meaningful data that `statfs(2)` can return to `df` for that mount point. Amd uses the “ignore” or “auto” mount type to tell the system to hide that mount entry from `df`. We found that this feature was difficult to test automatically: some systems do not define these mount types in their header files but still use them (hard-coded in the vendor’s own tools). Other systems define both, but prefer one over the other. A few systems define one or more of them but use an entirely different mechanism to hide mount entries: a mount table flag. Lastly, some systems do not have the ability to hide mount entries. We wrote the test simply and statically as follows:

```
case "${host_os}" in
  irix* | hpux10* )
    ac_cv_hide_mount_type="ignore" ;;
  sunos4* )
    ac_cv_hide_mount_type="auto" ;;
  * )
    ac_cv_hide_mount_type="nfs" ;;
esac
```

3.2 Automake

Automake [6] can automatically generate any number of Makefile templates for use when configuring packages. These Makefile templates contain the exact definitions for compiling, linking, installing programs and auxiliary files, and more. Automake allows for many features tested during the process of configuring the package to be passed on to Makefile templates. These Makefile templates are used by the `configure` script at configure time; the script performs simple variable substitution on the templates, to produce the actual Makefiles used to compile the package. The latter are the final Makefiles for that specifically-configured package and include any additional site overrides.

One example of Automake’s usefulness is that it creates templates that contain generic definitions for the libraries that applications need to link with. These libraries are detected early in the configuration process. This way the exact set of libraries needed is used during

build time. The names and locations of these libraries do not have to be statically configured or specified by the user.

One disadvantage of Automake is that it produces long and complex Makefile templates. These are more difficult to debug and understand because of their length and the large number of Makefile features they use which are intended to assist `configure` in producing a final Makefile.

To use Automake, a package maintainer writes small `Makefile.am` Automake template files that define the bare minimum that needs to be known at that point; this is often just the names of target files and the sources used to produce those targets. Then, the maintainer calls a small number of special M4 macros in their `configure.in` file which tell Autoconf that this package uses Automake-generated Makefiles. Next, the maintainer runs `automake` to produce the Makefile template files named `Makefile.in`. The `configure` script reads `Makefile.in` files at configure time and generates the final Makefiles used to compile the package.

3.3 Libtool

Libtool [7] automates the building, linking, and installation of shared or static libraries. Shared library support is specific to a given system. Different compiler, linker, and assembler options are often used to build shared libraries and those options depend on the specific development tools used. Some shared libraries use different extensions such as `.so` or `.sl`. Different systems use different rules for versioning of shared libraries. Some other systems require setting environment variables such as `$LD_LIBRARY_PATH` in order to run a binary with the proper shared library.

To use Libtool, a package maintainer calls a small number of special M4 macros in their `configure.in` file. Also, the maintainer must use a slightly different way of specifying libraries in various `Makefile.am` files, to indicate to Autoconf and Automake that this package will use Libtool to support both shared and static libraries.

4 Evaluation

When a package uses autotools such as Autoconf, it generally becomes easier to maintain. However, even Autoconf has its limitations. The first goal of this section is to provide a method of evaluating a package’s complexity for developers who are using or considering using autotools for that package. Note that we are specifically concerned with complexity concerning the portability of

that package to newer operating environments. The second goal of this section is to show the benefits and limitations of using autotools.

Table 1 lists the packages that we evaluated. We picked ten large, popular packages that use autotools, including Am-utils. We also evaluate the Amd package, which is Am-utils before it was autotooled. We evaluated as many versions of these packages as we could find, spanning development cycles of 2 to 9 years. This ensures that our reported results are sufficiently stable, given a large number of versions spanning several years.

Package	Versions	Year-span
amd	10	2.6
am-utils	48	4.7
bash	9	4.6
bin-utils	9	4.9
emacs	12	5.1
gcc	11	9.1
gdb	4	4.1
glibc	11	5.2
openssh	58	2.0
tcl	38	8.7
tk	37	8.1

Table 1: The packages evaluated in this section, the number of versions of each package we evaluated, and the overall span of release years for those versions.

Figures 1 through 6, include “error” bars showing one standard deviation off of the mean. Since we have evaluated a number of packages and versions for each, the standard deviation accounts for the general variation in size and complexity of the package over time.

4.1 Code Complexity

A typical metric of code complexity is a count of the number of lines of code in the package, as can be seen in Figure 1. As we can see, the four largest packages are Binutils, Gcc, Gdb, and Glibc. The number of lines of code in a package is one useful measure of the effort involved in developing and maintaining the package, but may not tell the whole story.

A different measure of the portability complexity of a package is the number of CPP conditionals that appear in the code: `#if`, `#ifdef`, `#ifndef`, `#else`, and `#elif`. Each of those statements indicates one extra code compilation diversion. Generally, each of those CPP conditionals account for some difference between systems, to ensure that the code can compile cleanly on each system. In other words, the effort to port a software

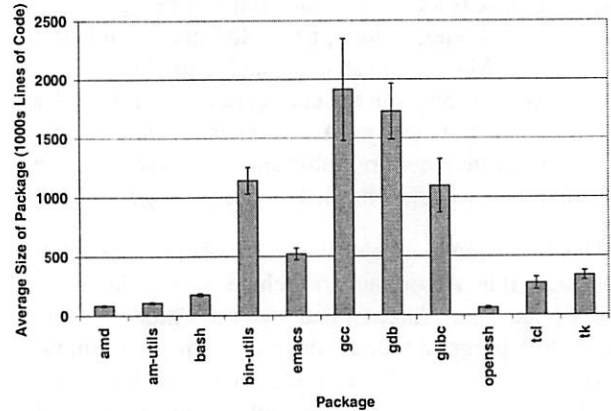


Figure 1: Average size of packages in thousands of lines of code.

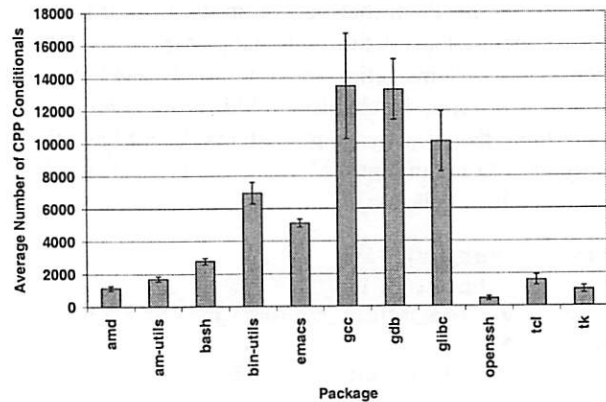


Figure 2: Average number of CPP conditionals per package.

package to multiple systems and maintain it on those is related to the number of CPP conditionals used.

Figure 2 shows the average number of CPP conditionals per package. Since Autoconf supports conditional whole source file compilations, we counted each of those conditionally-compiled source files as one additional CPP conditional. Here, the same packages that have the most lines of code (Figure 1) also have the most number of CPP conditionals. This is not surprising: as the size of the package grows, so the number of CPP conditionals is likely to grow. This suggests that perhaps neither code size nor absolute number of CPP conditionals provide a good measure of code complexity.

Next, we combined the above two metrics to provide a normalized metric of code complexity for the purposes of portability and maintainability of code over multiple operating systems. In Figure 3 we show the average number of CPP conditionals per 1000 lines of code. We notice three things in Figure 3.

First, the difference between the most and least com-

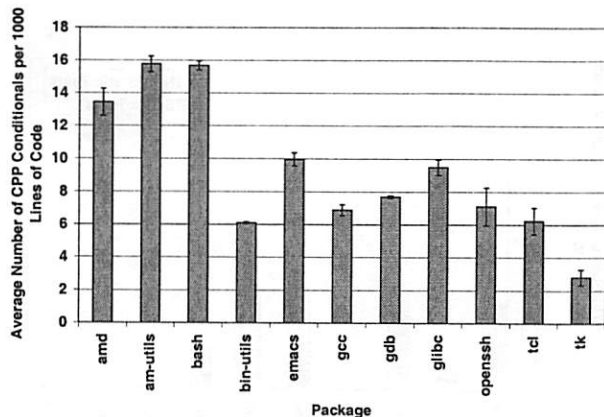


Figure 3: Average number of CPP conditionals per 1000 lines of code.

plex packages is not as large as in Figures 1 and 2. In those two, the difference was as much as an order of magnitude, whereas in Figure 3 the difference is a factor of 2–4.

Second, the standard deviation in this figure is smaller than in the first two. This means that although larger packages do have more CPP conditionals, the average distribution of CPP conditionals is almost fixed for a given package. The implication of this is also that a given package may have a native (portability) complexity that is not likely to change much over time and that this measure is related to the nature of the package, not its size.

The third and most important thing we notice in Figure 3 is that the packages that appear to be very complex in Figures 1 and 2 are no longer the most complex; leading in average distribution of CPP conditionals are Am-utils and Bash. To understand why, we have to understand what makes a software package more complex to port to another operating system. For the most part, languages such as C and C++ are portable across most systems. The biggest differences come when a C program begins interacting with the operating system and the system libraries, primarily through system calls. Although POSIX provides a common set of system call APIs [3], not all systems are POSIX compliant and every system includes many additional system calls and `ioctl`s that are not standardized. Furthermore, although the C library (`libc`) provides a common set of functions, many functions in it and in other libraries are not standardized. For example, there are no widely-standardized methods for accessing configuration files that often reside in `/etc`. Similarly, software (e.g., `libbfd` in Binutils) that handles different binary formats (ELF, COFF, `a.out`) must function properly across many platforms regardless of the binary formats supported by those platforms.

Despite their large size, Binutils, Gcc, Gdb, and Glibc—on average—do not use as many operating system features as Bash and Am-utils do. For example, Gcc and Binutils primarily need to be able to read files, process them internally (parsing, linking, etc.), and then write output files. Most of their complexity exists in portable C code that performs file parsing and target format generation. Whereas Gcc and Binutils are large and complex packages in their own right, *porting* them to other operating systems may not be as difficult a job as for a program that uses a wide variety of system calls or a program that interacts more closely with other parts of the running system. (In this paper we do not account separately for package-specific portability complexities: Gcc and Binutils to new architectures, Am-utils to new file systems, Emacs and Tk to new windowing systems, etc.)

For example, Bash must perform complex process and terminal management, and it invokes many system calls as well as special-purpose `ioctl`s. Amd (part of Am-utils) is a user-level file server and interacts heavily with the rest of the operating system to manage other file systems: it interacts with many local and remote services (NFS, NIS/NIS+, DNS, LDAP, Hesiod); it understands custom file systems (e.g., loop-device mounts in Linux, Cachefs on Solaris, XFS on IRIX, Autofs, and many more); it is both an NFS client and a local NFS server; and it communicates with the local host's kernel using an asynchronous RPC engine. By all rights, an automounter such as Amd is a file system server and *should* reside in the kernel. Indeed, Autofs [1, 4] is an effort to move the critical parts of the automounter into the kernel.

4.2 Autoconf Tests

Before building a package, it must be configured. The number of Autoconf (and Automake and Libtool) tests that a package must perform is another useful measure of the complexity of the package. The more tests performed, the more complex the package is to port to another system, since the package requires a larger number of system-discriminating features.

Figure 4 shows the average number of tests that each package performs. The figure validates some of what we already knew: that Binutils, Gcc, Gdb are large and complex. But we also see that Am-utils performs more than 600 tests: only Gdb performs more tests on average. This confirms that Am-utils is indeed a complex package to port, even though its size is more than ten times smaller than Gcc or Gdb.

Since Autoconf comes with many useful tests already, we also measured how many new Autoconf tests the

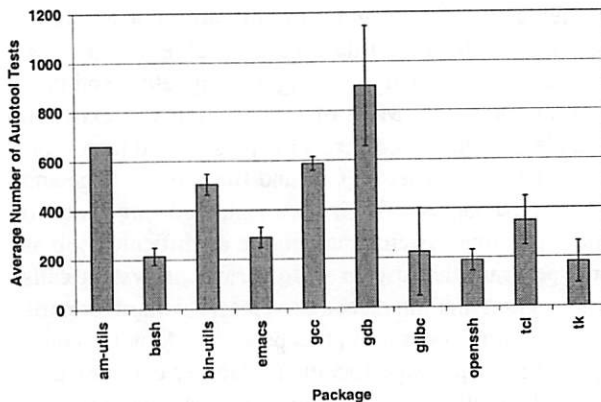


Figure 4: Average number of autotool tests (Autoconf, Automake, and Libtool M4 tests). The large standard deviation for Gcc is due to the fact that Gcc 2.x used a few canned configurations, whereas Gcc 3.x began using many Autoconf tests. Amd is excluded because it is the Am-utils package before autotooling, and hence includes no Autoconf tests.

package's maintainers wrote. The number of new macros that are written shows two things. First, that Autoconf is limited to what it already supports and that new macros are almost always needed for large packages. Second, that the number of new macros needed indicates that a given package may be more complex.

Figure 5 shows the average number of new M4 macros (Autoconf tests) that were written for each package. As we see, most packages do not need more than 10 new tests, if any. Binutils, Gcc, and Gdb are more complex and required about 30 new tests each. Am-utils, on the other hand, required nearly 90 new tests. According to this metric, Am-utils is more complex than the other packages we measured because it requires more tests that Autoconf does not provide. Indeed this has been true in our experience developing and maintaining Am-utils: many tests we wrote try to detect kernel features and internal behavior of certain system calls that none of the other packages deal with (for example, how to pass file-system-specific mount options to the mount (2) system call).

Figure 5 also shows the portion of static macros that were written. As we described in Section 3.1.4, these are macros that cannot detect a feature 100% deterministically and are often written as a case statement for different operating systems. In other words, these tests cannot use the power of Autoconf to perform automated feature detection. As we see in Figure 5, most packages need a few such static macros, if any. Again, Am-utils takes the lead on such static macros. The main reason for this is that a reliable way to test such features is impossible without superuser privileges and knowledge of the entire

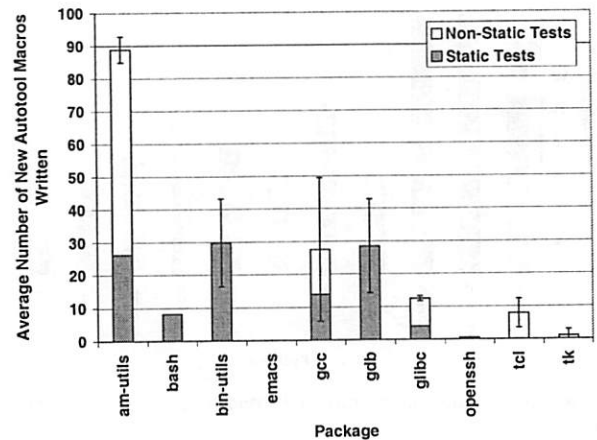


Figure 5: Average number of new Autoconf tests written and the portion of those that are static tests. Amd is excluded because it is the Am-utils package before autotooling, and hence includes no Autoconf tests.

site (i.e., the names, IP addresses, and exported resources of various network servers).

The conclusion we draw in this section is that although Autoconf continues to evolve and provide more tests, maintainers of large and complex packages may still have to write 10–30 custom macros. Moreover, some packages will always need a number of static macros, for those features that Autoconf cannot test in a reliable, automated way.

4.3 Amd and Am-utils

In Sections 4.1 and 4.2 we established that Am-utils is a more complex package to port than it appears from looking purely at its size. In several ways, Am-utils represents an upper bound for the complexity of portable C code: of the packages examined it has the most dense distribution of CPP conditionals and the largest number of custom macros. In addition, it performs critical file system services that are often part of the kernel proper. Therefore, analyzing Am-utils in more detail provides more insight into the process of maintaining and porting large or complex packages. Moreover, since we have maintained this code for nearly ten years, we can provide a unique perspective on the history of the development of Am-utils dating back to well before it used autotools.

In Figure 6 we see the code size for all released versions of Am-utils. The vertical dashed line separates the autotooled code on the right from the non-autotooled one on the left (before autotooling, the package had a “upl” versioning scheme). The most important factor is the drop in code size after autotooling. The last version before autotooling (amd-upl102) was 91640 lines long.

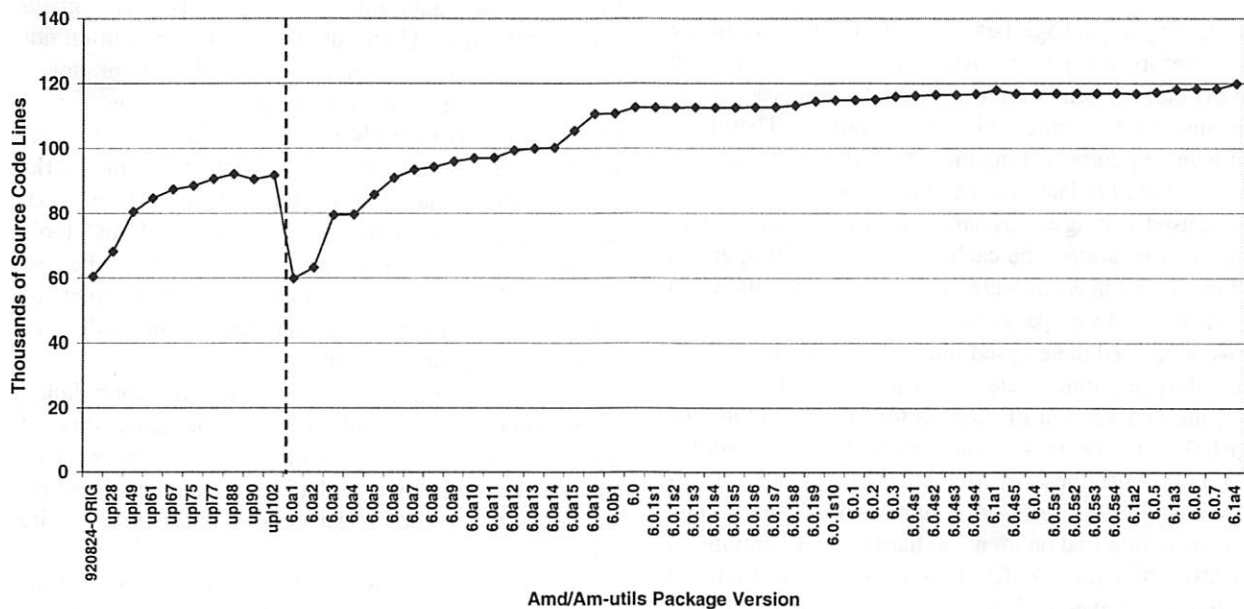


Figure 6: Code size of all Amd and Am-utils package versions. Versions to the right of the vertical line were autotooled.

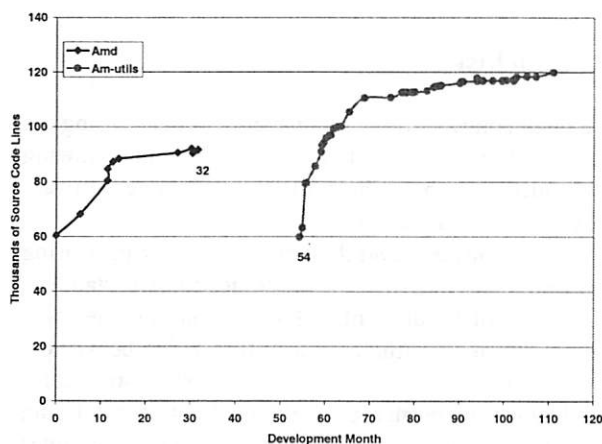


Figure 7: Development timeline of Amd and Am-utils, spanning nearly 10 years. There is a dramatic increase in code size (and hence features) for Am-utils in the first year after the package had been autotooled.

The first version after autotooling (am-utils-6.0a1) was only 59804 lines long. The two versions offered identical features and behavior, but the latter used Autoconf, Automake, and Libtool. This drop of 35% in code size was afforded thanks to the code cleanup and simplification that resulted from using the GNU autotools.

Since the release of many versions occurred over a period of nearly ten years, we show in Figure 7 the timeline for each release and the code size since the first release. We can see that for the first 32 months, nearly three years, the non-autotooled package continued to grow in size.

However, that growth in size was accompanied by a serious increase in difficulty to maintain the package which hindered the package's evolution; the older code contained a large number of multi-level nested CPP macros, often resulting in what is dubbed "spaghetti code." It took nearly two more years before the first autotooled version of Amd was released. Of those 22 months, about 6 months prior to month 54 were spent learning how to use Autoconf, Automake, and Libtool, understanding the inner working of Amd, and adapting the code to use autotools.²

The autotooling effort paid off significantly in two ways. First, the autotooled version was more than one-third smaller. Second, the autotooling process allowed us to add new features to Amd and port it to new systems with minimal effort. For the first 12 months after its initial autotooled release, Am-utils grew by 70%, adding major features such as NFSv3 [8] support, Autofs [1] support, a run-time automounter configuration file `/etc/amd.conf`, and offering dozens of new ports. This growth in size did not complicate the code much, as can be seen from the small standard deviation for Am-utils in Figure 3. The growth rate has reduced over the past two years, as the Am-utils package became more stable and fewer new features or ports were required.

The conclusion we draw from our experiences is that large packages can benefit greatly from using autotools such as Autoconf. Autoconf-based packages are easier to maintain, develop, and port to new systems.

4.4 Performance

Autotooling a package bears a build-time performance cost. Before compiling a package, `configure` must run to detect system features. This detection process can consume a lot of time and CPU resources. Therefore, Autoconf supports caching the results of a `configure` run, to be used in later invocations. These cached results can be used as long as no changes are made to the system that could invalidate the cache, such as an OS upgrade, installation of new software packages, or de-installation of existing software packages.

We measured the elapsed time it took to build a package before and immediately after it was autotooled. We used the last version of Amd before it was autotooled (`upl102`) and the first version after it was autotooled (`6.0a1`) because these two included functionally identical code. We ran tests on a number of different Unix systems configured on identical hardware: a Pentium-III 650MHz PC with 128MB of memory. We ran each test ten times and averaged the results. The standard deviations for these tests were less than 3% of the mean.

Action and Package	Time
Build Amd-upl102	35.4
Configure Am-utils-6.0a1 (no cache)	102.6
Configure Am-utils-6.0a1 (with cache)	24.2
Compile Am-utils-6.0a1	73.1

Table 2: Time (seconds) to Configure and Build Amd Packages

Table 2 shows the results of our tests. We see that just compiling the newly autotooled code takes more than twice as long. That is because the autotooled code includes long automatically generated header files such as `config.h` in every source file—despite the fact that the autotooling process reduced the size of the package itself by one-third. Worse, configuring the newer Am-utils package alone now takes more than 100 seconds. However, after that first run, re-configuring the package with cached results runs four times faster.

If we consider the worst-case overall time it takes to build this package, including the configuration part without a cache, then building Am-utils-6.0a1 is nearly five times slower than its functionally-equivalent predecessor, Amd-upl102.

4.5 Autotool Limitations

Through this work, we identified five limitations to GNU autotools. First, developers must be fairly knowledgeable in using these autotools, including understanding how they work internally.

Second, building code that was autotooled often takes longer than non-autotooled equivalents. However, given ever-increasing CPU speeds, this limitation is often not as important as ease of maintenance and configuration.

Third, developing code with autotools requires using a GNU version of the M4 processor. Also, Autoconf depends on the native system to provide stable and working versions of the Bourne shell `sh`, as well as `sed`, `cpp`, and `egrep` among others. Even though such tools come with most Unix systems, they do not always behave the same. When they behave differently, maintainers of GNU autotools must use common features that will work portably across all known systems.

Fourth, although autotools support cross-compilation environments which further helps the portability of cross-compiled code, Autoconf generally cannot execute binary tests meant for one platform on another. This limits Autoconf's ability to detect certain tests that require the execution of binaries.

Last, developers may still have to write custom tests and M4 macros for complex or large packages. Developing, testing, and debugging such tests is often difficult since they intermix M4 and shell syntax.

5 Conclusions

The main contribution of this paper is in quantifying the benefits of autotools such as GNU Autoconf: showing how much they help the portability of large software packages and illustrating their limitations.

We also showed several useful metrics for measuring the complexity of code when it comes to its portability: the average distribution of CPP conditionals in the code, the number of new Autoconf tests that had to be written, and the portion of those tests that were static and therefore beyond the normal capabilities of Autoconf. Using these metrics we showed how packages with more lines of code may not be as difficult to port as packages that use a more diverse set of system features.

In analyzing the evolution of the Am-utils package, we showed how the package benefited from autotooling: its code size was reduced dramatically and the code base was made cleaner, thus allowing rapid progress on new feature implementation.

The GNU autotools we used also have limitations. First, maintainers must become intimately familiar with the autotools. Second, building autotooled packages takes longer. Third, even autotools cannot solve all portability problems; large-package maintainers usually have to write their own custom tests. Fourth, certain tests cannot be executed in a cross-compiled environment. Nevertheless, in the long run, once an initial autotooling ef-

fort has taken place, an autotooled package is easier to maintain on existing systems and port to new or diverging systems.

Although a count of code lines had been used for years as a metric of code complexity, we believe that metric oversimplifies the process of portable software development. In the future we would like to automate the process of quantifying the complexity of a package. We plan on building tools that will parse autotool files and related C code, separately evaluating conditionally-included code from unconditionally-included code. This will allow us to evaluate how much of a given autotooled package is highly portable code and how much code is densely populated with system-specific code. In addition, we would like to account for package-specific portability complexities (i.e., how new architectures affect Gcc and Binutils, new file systems affect Amd, and new windowing systems affect Emacs and Tk).

6 Acknowledgments

We would like to thank the many members of the Am-utils developer's list and our co-maintainers for their contributions to the code, recommendations, and comments. Many of them provided us remote and superuser access to different systems so we could compile and test new code on as many platforms as possible. Several people made significant contributions to the code: Ion Bădulescu implemented Autofs support and maintains the Linux support in Am-utils, despite the increase in divergence among Linux systems; Randall S. Winchester and Christos Zoulas contributed the initial NFS V.3 code.

Additional thanks go to the anonymous Usenix reviewers and especially Chuck Cranor, Chris Demetriou, and Niels Provos for their valuable comments. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374, as well as by an HP/Intel gift number 87128.

To retrieve the Am-utils software, including documentation and the dozens of new M4 macros written for Am-utils, visit <http://www.am-utils.org>.

References

- [1] B. Callaghan and S. Singh. The autofs automounter. In *Proceedings of the Summer USENIX Technical Conference*, pages 59–68, Summer 1993.
- [2] J. S. Haemer. Imake rhymes with mistake. *;login:*, 19(1):32–3, Jan-Feb 1994.
- [3] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Technical Report STD-1003.1, ISO/IEC, 1996.
- [4] R. Labiaga. Enhancements to the Autofs Automounter. In *Proceedings of the Thirteenth USENIX Systems Administration Conference (LISA '99)*, pages 165–174, Seattle, WA, November 1999.
- [5] D. MacKenzie. Autoconf: Creating automatic configuration scripts. Technical report, FSF, 1996.
- [6] D. MacKenzie and T. Tromey. Gnu automake. Technical report, FSF, 1997.
- [7] G. Matzigkeit. Gnu libtool. Technical report, FSF, 1997.
- [8] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, June 1994.
- [9] J. S. Pendry and N. Williams. *Amd – The 4.4 BSD Automounter*, 5.3 alpha edition, March 1991.
- [10] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, October 2000.
- [11] L. Wall, H. Stenn, and R. Manfredi. dist-3.0. Technical report, 1997. <ftp.funet.fi/pub/languages/perl/CPAN/authors/id/RAM/>.
- [12] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.

Notes

¹A similar test to detect names of fields within structures was recently added to Autoconf, partially based on our efforts in writing and contributing M4 test macros to the OSS community.

²Amd was originally written by Jan-Simon Pendry and Nick Williams, not the authors of this paper. Therefore we did not initially understand every part of the original 60432-line code base.

Simple Memory Protection for Embedded Operating System Kernels

Frank W. Miller *

Department of Computer Science & Electrical Engineering
University of Maryland, Baltimore County

Abstract

This work describes the design and implementation of memory protection in the *Roadrunner* operating system. The design is portable between various CPUs that provide page-level protection using Memory-Management Unit (MMU) hardware. The approach overlays protection domains on regions of physical memory that are in use by application processes and the operating system kernel. An analysis of code size shows that this design and implementation can be executed with an order of magnitude less code than that of implementations providing separate address spaces.

1 Introduction

This work presents the design and implementation of a memory protection subsystem for an operating system kernel. It may be useful for systems that do not or cannot utilize paging and/or swapping of memory to secondary storage. The types of computer systems in use today that have such a requirement are generally embedded. The primary goals of the design and implementation are simplicity and small code size.

There are a variety of commercial and academic embedded operating system kernels available today. Many do not implement memory protection of any kind. This work presents a general,

page-based mechanism that can be used to add memory protection to these kernels. The initial design and implementation was performed in the *Roadrunner* operating system kernel, an embedded kernel similar to the VxWorks and pSOS kernels.

Virtually all modern operating system kernels that make use of MMU hardware utilize the address translation mechanism to implement virtual addressing and separate address spaces. The notable exceptions have been embedded operating system kernels. Bypassing address translation and making use strictly of the memory protection mechanisms provided by MMU hardware yields a simple memory protection design that can be implemented with an order of magnitude less code than designs that provide separate virtual address spaces. This code reduction is achieved even when a comparison is done excluding the code for paging and swapping.

The design uses page-based memory protection to 1) limit access by one process to the memory regions of other processes and 2) limit access by user processes to kernel memory. It does not utilize separate virtual address spaces or provide the illusion that physical memory is larger than it really is by using demand-paging and/or swapping.

The design provides protection portably. It can be implemented on a variety of different MMU hardware devices and the design implications associated with several MMUs are discussed.

The remainder of this work is organized as fol-

*Author's current address: *sentilO Networks, Inc.*, 2096 Gaither Road, Rockville, Maryland 20850
Email:fwmiller@cornfed.com

lows. Section 2 provides an introduction to the *Roadrunner* operating system in which the new memory protection mechanism is implemented. Section 3 provides the specifics of the design and implementation of the memory protection subsystem. Section 4 discusses the design points surrounding MMU designs found in three popular CPUs. Section 5 presents a set of measurements yielded by the initial implementation of this protection scheme in the *Roadrunner* operating system. Section 6 provides a brief survey of related work and Section 7 draws the work to its conclusion.

2 *Roadrunner* Design

There are three basic *Roadrunner* abstractions:

Processes: The basic unit of execution is the process. The *Roadrunner* kernel provides a POSIX-like process model for concurrency.

Files: A variety of I/O operations, including inter-process communications and device access are implemented using a multi-layered file system design.

Sockets: An implementation of the Internet protocols is available through a BSD Sockets interface.

These abstractions have an Application Programming Interface (API) that is exported through the kernel system call interface.

An important distinction between this design approach and that of separate address spaces is that user code must either be position-independent or have relocations performed at load time. The *Roadrunner* implementation performs relocation by default so position-independent code (PIC) is not required.

Memory Protection Operations

This work describes the design and implementation of the memory protection subsystem present in the *Roadrunner* operating system kernel [2, 4]. The basic design principle is that logical (or virtual) addresses are mapped one-to-one

to physical addresses. This means that the value of logical address is the same as its corresponding physical address and that all processes reside in the same logical as well as physical address space. One-to-one mapping simplifies the memory management subsystem design dramatically. In addition, when the indirection represented by address translation is removed, the programmer can reason about the actual logical addresses as physical addresses and that can be useful for dealing with memory-mapped elements.

Protection is based on *domains*. A domain is a set of memory pages that are mapped using a set of page tables. In addition, a set of memory pages associated with the operating system kernel called the *kernel map* is kept. The kernel map is mapped into all domains but is accessible only in supervisor mode.

No address translation is performed. Only the protections attributes associated with page table entries are used. The basic operations that can be performed are listed as:

- Insert the mapping of a page into a domain
- Remove the mapping of a page from a domain
- Insert the mapping of a page into the kernel map
- Remove the mapping of a page from the kernel map
- Update a domain to reflect the current mappings in the kernel map

Table 1 lists the routines in the *Roadrunner* kernel that implement these basic operations.

3 The *Roadrunner* Implementation of Memory Protection

There are three basic memory management data structures used in *Roadrunner*:

1. *Region Table*: an array of regions that track the allocations of all the physical memory in the system
2. *Page Tables*: each process belongs to a protection domain that is defined by the page tables that are associated with that process
3. *Kernel Map*: a list of mappings that are entered into all the page tables in the system but are accessible only when the CPU is in supervisor mode

Table 1: Page table management routines

<code>vm_map(pt, page, attr)</code>	Map a single page into a set of page tables
<code>vm_map_range(pt, start, len, attr)</code>	Map a sequence of contiguous pages into a set of page tables
<code>vm_unmap(pt, page)</code>	Remove the mapping for a single page from a set of page tables
<code>vm_unmap_range(pt, start, len)</code>	Remove the mapping for a contiguous sequence of pages from a set of page tables
<code>vm_kmap_insert(entry)</code>	Insert a sequence of pages into the kernel map
<code>vm_kmap_remove(entry)</code>	Remove a sequence of pages from the kernel map
<code>vm_kmap(pt)</code>	Update specified page tables to reflect the current mappings in the kernel map

These three data structures are used in conjunction by the kernel memory management routines that are exported for use by the rest of the kernel subsystems and by user applications through the system call interface.

Regions

The basic unit of memory allocation is the *region*. A region is defined as a page-aligned, contiguous sequence of addressable locations that are tracked by a starting address and a length that must be a multiple of the page size. The entire physical memory on a given machine is managed using a boundary-tag heap implementation in *Roadrunner*. Figure 1 illustrates the basic data structure used to track the allocations of memory regions. Each region is tracked using its starting address, `start`, and length, `len`. Each region is owned by the process that allocated it originally, or by a process to which ownership has been transferred after allocation. The `proc` field tracks which process currently owns the region. Two double-linked lists of region data structures are maintained, using the `prev` and `next` fields, each in ascending order of starting address. The first is the free list, those regions that are not allocated to any process.

The second is the allocated list, those regions that are being used by some process.

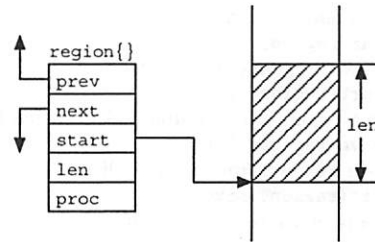


Figure 1: A region on one of the heap lists

Table 2 lists the routines used to manage the heap. The `valid_region()` routine provides a check for whether a pointer, specified by the `start` parameter, corresponds to the starting address of some region. It also serves as a general lookup routine for locating a region data structure given a starting address. The rest of the routines take a pointer to a region data structure like the one illustrated in Figure 1 as their first parameter. The `region_clear()` routine sets the fields of a region data structure to initialized values. The `region_insert()` routine inserts a region in ascending starting address order into a double-linked region list, specified by the `list` parameter. This routine is used to insert a region into either the free or allocated region lists. The `region_remove()` routine removes a region from the specified list. The `region_split()` routine takes one region and splits it into two regions. The `size` parameter specifies the offset from the beginning of the original region where the split is to occur.

Page Tables

The kernel keeps track of the page tables present in the system by maintaining a list of page table records. Figure 2 illustrates the page table record data structure and the associated page tables. The page table records are kept in a single-linked list using the `next` field. If multiple threads are executed within a single protection

Table 2: Region management routines

<code>valid_region(start)</code>	Check whether pointer corresponds to the starting address of a region
<code>region_clear(region)</code>	Initialize a region data structure
<code>region_insert(region, list)</code>	Insert a region into a double-linked region list
<code>region_remove(region, list)</code>	Remove a region from a region list
<code>region_split(region, size)</code>	Split a region into two regions

domain, the `refcnt` field tracks the total number of threads within the domain. The `pd` field points to the actual page table data structures. Note that there is a single pointer to the page tables themselves. This design implies that the page tables are arranged contiguously in memory. An assessment of current MMU implementations in several popular CPU architectures indicates that this is a reasonable assumption. More details on the page table structures of several popular processor architectures are given in Section 4.

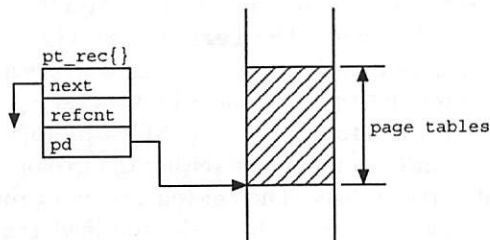


Figure 2: A page table record and its associated page tables

The first four routines in Table 1 implement the basic protection mechanism. They enter and remove address mappings to and from page tables, respectively. All four of these routines operate on a set of page tables specified by their first parameter, `pt`. The `vm_map()` routine provides the fundamental operation of inserting a mapping for a single page into a set of page tables. The page found at the location specified by the `start`

parameter is inserted with the protection attributes specified by the `attr` parameter into the specified page tables. `vm_map_range()` is provided for convenience as a front-end to `vm_map()` to allow mapping a sequence of contiguous pages with a single call. The `start` parameter specifies the address of the first of a contiguous sequence of pages. The `len` parameter specifies the length, in bytes, of the page sequence. The initial implementation or the `vm_map_range()` routine makes calls to `vm_map()` for each page in the specified range. This implementation is obviously ripe for optimization.

The `vm_unmap()` routine balances `vm_map()` by providing the removal of a single page mapping from a page table. The `page` parameter specifies the starting address of the page that is to be unmapped. `vm_unmap_range()` is provided as a front-end to `vm_unmap()` to allow removal of a sequence of contiguous entries with a single call. `start` specifies the starting address of the page sequence and `len` gives the byte length of the page sequence to be unmapped. The `vm_unmap_range()` routine also make individual calls to `vm_unmap()` for each page in the specified range and can also be optimized.

The Kernel Map

In some virtual memory system designs that provide separate address spaces, the kernel has been maintained in its own address space. In the *Roadrunner* system, the memory used to hold the kernel and its associated data structures are mapped into all the page tables in the system. Kernel memory protection is provided by making these pages accessible only when the CPU has entered supervisor mode and that happens only when an interrupt occurs or a system call is made. The result is that system calls require only a transition from user to supervisor mode rather than a full context switch.

The kernel map is an array of kernel map entries where each entry represents a region that is entered in the kernel map. Figure 3 illustrates the structure of one of these kernel map entries and the region of memory that it represents. The

`start` and `len` fields track the starting address and length of the region. The `attr` field stores the attributes that are associated with the pages in the region. This information is used when the pages are entered into a set of page tables by the `vm_kmap()` routine.

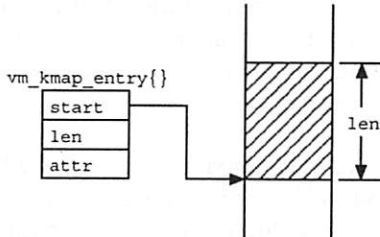


Figure 3: A kernel map entry and its associated memory region

The last three routines in Table 1 provide the API for managing the kernel map. The `vm_kmap_insert()` routine enters a kernel map entry, specified by the `entry` parameter, into the kernel map. The `vm_kmap_remove()` routine removes a previously entered kernel map entry, also specified by the `entry` parameter, from the kernel map. The `vm_kmap()` routine causes a set of page tables, specified by the `pt` parameter, to be updated with the current kernel map entries.

Page Faults

The most important function of page faults in a system using separate virtual address spaces is demand paging. Demand paging of user code can also be done using this approach under two additional conditions. First, all of the physical memory required to hold the program must be allocated when the program is started. Second, code relocation needs to be performed on-the-fly when sections of the program were loaded on-demand.

If demand paging of user code is implemented, page fault handling is similar to systems where separate virtual address spaces are used. When

a page fault occurs, the appropriate kernel service determines whether the fault occurred due to a code reference and if so, it loads the appropriated section of code and restarts the faulting process.

The initial *Roadrunner* implementation does not currently support demand paging of program code. As such, page fault handling is trivial, resulting in the termination of the process that caused the fault.

Kernel Memory Management

Table 3 lists the routines that are used by kernel subsystems and by applications through the system call interface to allocate and free memory from the global heap.

Table 3: Kernel memory management routines

malloc(size)	Allocate a region of memory in the calling process's protection domain
free(start)	Free a region of memory previously allocated to the calling process
kmalloc(size)	Allocate a region of memory for the kernel
kfree(start)	Free a region of memory previously allocated to the kernel

The `malloc()` routine performs an allocation on behalf of a process by performing a first-fit search of the free list. When a region is found that is at least as large as a request specified by the `size` parameter, it is removed from the free list using the `region_remove()` routine. The remainder is split off using the `region_split()` routine and returned to the free list using the `region_insert()` routine. The region satisfying the request is then mapped into the protection domain of the calling process using the `vm_map_range()` routine.

The `free()` routine returns a previously allocated region to the heap. After obtaining the region corresponding to the specified `start` parameter using the `valid_region()` lookup, the

`region_insert()` routine is used to enter the region into the free list. The inserted region is then merged with its neighbors, both previous and next if they are adjacent. Adjacency means that the two regions together form a contiguous sequence of pages. Merging is done to reduce fragmentation.

The `kmalloc()` routine allocates some memory on behalf of the kernel. After obtaining a region from the heap in a manner similar to the `malloc()` routine based on the specified size request, an entry is placed into the kernel map using the `vm_kmap_insert()` routine. This action records the new region as an element of the kernel map. Subsequent calls to `vm_kmap()` will cause the new region to be accessible as part of the kernel when a process is running in supervisor mode.

The `kfree()` routine first removes the kernel mapping for the region specified by the `start` parameter using `vm_kmap_remove()`. The region is then placed back on the free list using `region_insert()`.

4 Assessing Various Memory Management Architectures

The *Roadrunner* memory protection system is designed to be portable. It makes few assumptions about the underlying hardware and can be ported to a variety of architectures. In this section, some of the details encountered when implementing this memory protection mechanism on several processors are presented. The initial implementation effort has focused on the IA-32 architecture but some design elements necessary for two other CPU architectures are also discussed.

These details are hidden by the interface given in Table 1. The interface to the memory protection subsystem remains the same when *Roadrunner* runs on different processors but the underlying implementation of the interface is different.

Intel IA-32

The initial implementation effort has been focused on the Intel IA-32 (or x86) Instruction Set Architecture (ISA) [3]. This architecture provides hardware support for both segmentation and paging. The segmentation features are bypassed completely. This is done by initializing all the segment registers to address the entire 4 Gbyte address space.

The hardware has a fixed page size of 4 Kbytes and a two-level hierarchical page table structure is used. The first level is a single page called the page directory. Each entry in the page directory contains a pointer to a second-level page table. Each entry in the page table contains a pointer to a target physical page.

Since the *Roadrunner* memory protection mechanism maps logical addresses one-to-one to physical addresses, the kernel need only maintain in the worst case, the page directory and enough pages to cover the physical memory in the machine. As an example, each second-level page table addresses 4 Mbytes of physical memory so a machine with 64 Mbytes of main memory requires 1 page for the page directory and 16 pages (or 64 Kbytes) for second-level page tables or a total of 17 total pages for each set of page tables.

The current implementation allocates page tables statically using this worst-case formulation. Future enhancements will provide demand-allocation of second-level page table pages. Demand-allocation occurs when a `vm_map()` operation is executed and the page directory indicates that no second-level page table has been allocated to handle the address range in which the specified address to be mapped falls.

Motorola PowerPC

The Motorola PowerPC CPU [5] is a Reduced-Instruction Set Computer (RISC) ISA that is popular in the embedded world. This architecture provides three hardware mechanisms for

memory protection, segmentation, Block Address Translation (BAT), and paging. Segmentation is bypassed in a manner similar to that of the IA-32 ISA. BAT is intended to provide address translations for ranges that are larger than a single page. This mechanism, included to allow addressing of hardware elements such as frame buffers and memory-mapped I/O devices, is also bypassed by initializing the BAT array such that no logical addresses match any array element.

The PowerPC also uses a 4 Kbyte page size but instead of a two-level hierarchy, implements a hashed page table organization. Some bits of the logical address are hashed to determine the page table entry that contains the corresponding physical address. Page Table Entries (PTEs) are organized into Page Table Entry Groups (PTEGs). There are two hashing functions, primary and secondary, that are used to determine which PTEG a logical address resides in. The primary hash function is applied first and the resulting PTEG is searched for the matching PTE. If the search fails, the secondary function is applied and a second PTEG is searched. If either search succeeds, the resulting PTE yields the corresponding physical address. If both searches fail, a page fault occurs.

Since the implementation of the page tables is irrelevant behind the generic interface, this organization does not provide any functional difference to that of the IA-32 two-level hierarchy. However, it does have implications for physical memory usage. For the hardware to locate a PTEG in main memory, all pages in the page tables must be laid out contiguously in physical memory. This constraint does not exist in the IA-32 design, i.e. the location of second-level page table pages can be arbitrarily placed in physical memory.

For separate, demand-paged, virtual address spaces, the hash table organization has the advantage of allowing the overall size of the page tables to be varied with respect to the desired hit and collision rates. Table 7-21 in [5] discusses the recommended minimums for page table sizes. In general, these sizes are larger than those required for the IA-32 CPU, e.g. 512

Kbytes of page table is recommended for a main memory size of 64 Mbytes. Note however, that these recommendations are tailored to providing a separate 32-bit logical address space to each process.

In the *Roadrunner* mechanism, a different calculation is required. For a given amount of main memory, the page table should be given a *maximum* size that allows all of the physical memory pages can be mapped simultaneously. This value is the same as the size quoted for the IA-32 implementation. In a manner analogous to demand-allocation of second-level page tables, the size of the hashed page table can be tailored dynamically to focus on the actual amount of memory allocated to a given process, rather than the entire physical memory.

MIPS

Another popular RISC ISA is the MIPS core [8]. There are a number of CPUs that are based on the MIPS ISA but they fall into two broad categories, based on either the MIPS R3000 and R4000 cores. There are variations on the specifics of the PTEs for these two cores but the basic principles are the same in both.

While the IA-32 and PowerPC architectures provide hardware mechanisms for loading a PTE into a Translation Lookaside Buffer (TLB) entry when a page fault occurs, the MIPS architecture requires the operating system kernel software to perform this function. If an address is asserted and a corresponding TLB entry is present, the physical address is acquired and used to access memory. If no matching TLB entry is present, a page fault is signaled to the operating system kernel.

The MIPS PTE is divided into three basic elements, the Address Space Identifier (ASID), the Virtual Page Number (VPN), and the offset within the page. In *Roadrunner*, the ASID can be used to reference the protection domain, the VPN references a page within the domain and the offset completes the physical address reference. The VPN is the upper bits of the logical

address and it is used as a key to lookup the corresponding physical page address in the page tables.

The MIPS architecture places no requirements on the page table structure in main memory. The operating system developer can tailor the page table structure as desired. In *Roadrunner*, a system targeted at resource-constrained applications, the goal is to minimize main memory usage.

Management of the TLB is explicit and since domains contain unique addresses, identifying entries to be discarded when the TLB is updated due to a context switch is made easier.

5 Measuring Memory Protection Performance

The major advantage of the memory protection design presented in this work is its simplicity. The *Roadrunner* design requires roughly an order of magnitude less code to implement the same function. The memory management code in the 2.2.14 Linux kernel consists of *approximately* 6689 lines of C found in the `/usr/src/linux/mm` and `/usr/src/linux/arch/i386/mm` directories as shipped with the Redhat 6.2 distribution [9]. The count for the Linux implementation explicitly excludes code that performs swapping. The *Roadrunner* memory protection implementation consists of 658 lines of C source code in a set of files found in the `roadrunner/sys/src/kern` directory and 74 lines of IA-32 assembly found in the `roadrunner/sys/src/kern/asm.S` source file.

The advantage of this approach would be unimportant if the performance of the design was unacceptable. A series of measurements presented in Table 4 demonstrate that this approach presents extremely good performance. The set of measurements was obtained using an Intel motherboard with a 1.7 GHz Pentium 4 CPU, 512 Kbytes of second-level cache, and 256 Mbytes of RDRAM. All measurements are the

average of a large number of samples taken using the Pentium timestamp counter, which runs at the resolution of the processor clock.

Table 4: Performance of *Roadrunner* Memory Protection Operations

Operation	Average Execution Time (μ sec)
<code>vm_map()</code>	0.15
<code>vm_map_range()</code>	4.22
<code>vm_unmap()</code>	0.11
<code>vm_unmap_range()</code>	0.41
<code>vm_kmap_insert()</code>	1.27
<code>vm_kmap()</code>	123.0
<code>malloc()</code>	1.45
<code>free()</code>	0.91
<code>kmalloc()</code>	3.24
<code>exec()</code>	1710.0
Context switch	1.71

The basic memory management routines presented in Table 1 provide the building blocks for other routines and as such, need to operate very quickly. The all-important `vm_map()` and `vm_unmap()` both exhibit execution times between 100 and 150 *nanoseconds*. Even with the naive implementation of `vm_map_range()` where each page table entry requires a call to `vm_map()`, an average of 256 pages (for these measurements) can be mapped in an average time of approximately four microseconds. Clearing page table entries is faster. Setting up entries requires several logic operations to set appropriate permissions bits. Clearing simply zeros entries yielding the approximately 410 *nanoseconds* to to clear the entries for an average of 196 pages (for these measurements).

The kernel memory management routines presented in Table 3 fall into two categories. `malloc()` and `free()` are called very often on behalf of applications. These routines need to perform better than the corresponding kernel routines, `kmalloc()` and `kfree()`, which are only used within the kernel to allocate data areas for kernel subsystems. All the routines have latencies between 1 and 2 microseconds except for `kmalloc()`. This operation is expensive since it requires a call to `vm_kmap_insert()`.

The latencies associated with `vm_kmap()` and `exec()` represent the most expensive operations presented here. The prior operations represents approximately 40 calls to `vm_unmap_range()`. The latter operation is a hybrid system call representing a combination of the semantics associated with `fork` and `execve` in typical UNIX-like systems. `exec()` loads a program and starts it running as a new process in its own protection domain. This operations requires a variety of initializations in addition to setting up the protection domain. The value measured here excludes the load time of the program code.

6 Related Work

The design of the *Roadrunner* memory protection subsystem and the structure of this paper were influenced most heavily by Rashid, *et. al.* in [7]. However, the Mach VM system provides separate, demand-paged, virtual address spaces as compared to the single address space in *Roadrunner*. Also, the *Roadrunner* design does away with the address map abstraction. The single address space allows the use of the generic interface given in Table 1, which is closer to the Mach pmap interface, to be substituted for the hardware-independent address map data structure and routines.

There is a significant body of work in the area of Single-Address-Space Operating Systems (SASOS) that is typified by the Opal system developed at the University of Washington [1]. These systems generally seek to provide a single *virtual* address space, that is, they still perform a translation between the logical and physical address.

The EMERALDS micro-kernel developed at the University of Michigan [10] is an example of a kernel designed specifically for the resource-constrained environments in small-to-medium sized embedded systems. The kernel includes a traditional multi-threaded, process-oriented concurrency model. The system maps the pages associated with the operating system kernel into the page tables of every process and uses the

user/supervisor transition to implement system calls in a manner that is analogous to the use of the kernel map in the *Roadrunner* design.

The Mythos microkernel [6] was developed as a threads based system to support the GNAT (Gnu Ada 9X translator) project. The kernel provided a Pthreads-based interface for concurrency to applications through the kernel API in a manner similar to the *Roadrunner* kernel. The system did not provide memory protection however.

7 Conclusion

This work has presented the design and implementation of a new memory protection mechanism that provides traditional levels of protection using significantly less code than designs that perform address translations. This design can be added to existing embedded operating system kernels that do not currently provide memory protection easily due to its small implementation effort. In addition, the design is portable among CPU architectures, which makes it even more attractive for use in these kernels, since they tend to be available for several different processors.

The *Roadrunner* operating system, in which the first implementation was performed, is free software, available under the GNU General Public License (GPL), on the World-Wide Web at <http://www.cornfed.com>.

References

- [1] Chase, J., *et. al.*, "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems*, 12, 4, 1994, pp. 271-307.
- [2] Cornfed Systems, Inc., *The Roadrunner Operating System*, 2000.
- [3] Intel Corp., *80386 Programmer's Reference Manual*, 1986.
- [4] Miller, F. W., "pk: A POSIX Threads Kernel", *FREENIX track, 1999 USENIX Annual Technical Conference*, 1999.

- [5] Motorola Inc., *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, 1997.
- [6] Mueller, F., Rustagi, V., and Baker, T., *Mythos - a micro-kernel threads operating system*, TR 94-11, Dept. of Computer Science, Florida State University, 1994.
- [7] Rashid, R., et. al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Proc. of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 1987.
- [8] Sweetman, D., *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.
- [9] Redhat, Inc., *Redhat Linux 6.2*.
- [10] Zuberi, K. M., Pillai, P., and Shin, K. G., "EMERALDS: a small-memory real-time microkernel", *17th ACM Symposium on Operating System Principles (SOSP99)*, 1999.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *login*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association

Interhack Corporation	Smart Storage, Inc.
Lucent Technologies	Sun Microsystems, Inc.
Microsoft Research	Sybase, Inc.
Motorola Australia Software Centre	Taos: The Sys Admin Company
The SANS Institute	TechTarget.com
Sendmail, Inc.	UUNET Technologies, Inc.

Supporting Members of SAGE

Certainty Solutions	New Riders Publishing
Collective Technologies	O'Reilly & Associates Inc.
ESM Services, Inc.	Ripe NCC
Lessing & Partner	Taos: The Sys Admin Company
Microsoft Research	Unix Guru Universe
Motorola Australia Software Centre	

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-880446-01-4